

SOFTWARE ENGINEERING LABORATORY SERIES

SEL-94-004

COLLECTED SOFTWARE ENGINEERING PAPERS: VOLUME XII

NOVEMBER 1994

(NASA-CR-189409) COLLECTED
SOFTWARE ENGINEERING PAPERS, VOLUME
12 (NASA. Goddard Space Flight
Center) 117 p

N95-28713

Unclas



National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

G3/61 0053148

**COLLECTED SOFTWARE
ENGINEERING PAPERS:
VOLUME XII**

NOVEMBER 1994



National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

Foreword

The Software Engineering Laboratory (SEL) is an organization sponsored by the National Aeronautics and Space Administration/Goddard Space Flight Center (NASA/GSFC) and created to investigate the effectiveness of software engineering technologies when applied to the development of applications software. The SEL was created in 1976 and has three primary organizational members:

NASA/GSFC, Software Engineering Branch

University of Maryland, Department of Computer Science

Computer Sciences Corporation, Software Engineering Operation

The goals of the SEL are (1) to understand the software development process in the GSFC environment; (2) to measure the effect of various methodologies, tools, and models on this process; and (3) to identify and then to apply successful development practices. The activities, findings, and recommendations of the SEL are recorded in the Software Engineering Laboratory Series, a continuing series of reports that includes this document.

Single copies of this document can be obtained by writing to

Software Engineering Branch
Code 552
Goddard Space Flight Center
Greenbelt, Maryland 20771

TABLE OF CONTENTS

Section 1—Introduction	1-1
Section 2—Software Measurement	2-1
“A Change Analysis Process to Characterize Software Maintenance Projects,” L. C. Briand, V. R. Basili, Y. Kim, and D. R. Squier	2-3
<i>Defining and Validating High-Level Design Metrics</i> , L. Briand, S. Morasca, and V. R. Basili	2-15
Section 3—Technology Evaluations	3-1
<i>Comparing Detection Methods for Software Requirements Inspections: A Replicated Experiment</i> , A. A. Porter, L. G. Votta Jr., and V. R. Basili	3-3
“Software Process Evolution at the SEL,” V. Basili and S. Green	3-33
Section 4—Ada Technology	4-1
“Genericity Versus Inheritance Reconsidered: Self-Reference Using Generics,” E. Seidewitz	4-3
Standard Bibliography of SEL Literature	

SECTION 1—INTRODUCTION

This document is a collection of selected technical papers produced by participants in the Software Engineering Laboratory (SEL) from November 1993 through October 1994. The purpose of the document is to make available, in one reference, some results of SEL research that originally appeared in a number of different forums. This is the 12th such volume of technical papers produced by the SEL. Although these papers cover several topics related to software engineering, they do not encompass the entire scope of SEL activities and interests. Additional information about the SEL and its research efforts may be obtained from the sources listed in the bibliography at the end of this document.

For the convenience of this presentation, the five papers contained here are grouped into three major sections:

- Software Measurement
- Technology Evaluations
- Ada Technology

The first section (Section 2) includes a study on the analysis of software maintenance changes to understand the flaws in the change process and a study on the comparison of four strategies for defining high-level design metrics. Section 3 presents studies on software inspection techniques and the SEL's Quality Improvement Paradigm. A study on simulating inheritance in an object-oriented environment appears in Section 4.

The SEL is actively working to understand and improve the software development process at Goddard Space Flight Center (GSFC). Future efforts will be documented in additional volumes of the *Collected Software Engineering Papers* and other SEL publications.

SECTION 2—SOFTWARE MEASUREMENT

The technical papers included in this section were originally prepared as indicated below.

- “A Change Analysis Process to Characterize Software Maintenance Projects,” L. C. Briand, V. R. Basili, Y. Kim, and D. R. Squier, *Proceedings of the International Conference on Software Maintenance*, September 1994
- *Defining and Validating High-Level Design Metrics*, L. Briand, S. Morasca, and V. R. Basili, University of Maryland, Technical Report TR-3301, June 1994

A Change Analysis Process to Characterize Software Maintenance Projects

Lionel C. Briand, Victor R. Basili, Yong-Mi Kim

Computer Science Department and Institute for Advanced Computer Studies
University of Maryland, College Park, MD, 20742

Donald R. Squier

Computer Sciences Corporation
System Sciences Division
Lanham-Seabrook, MD, 20706

Abstract

In order to improve software maintenance processes, we need to be able to first characterize and assess them. This task needs to be performed in depth and with objectivity since the problems are complex. One approach is to set up a measurement program specifically aimed at maintenance. However, establishing a measurement program requires that one understands the issues and is able to characterize the maintenance environment and processes in order to collect suitable and cost-effective data. Also, enacting such a program and getting usable data sets takes time. A short term substitute is needed.

We propose in this paper a characterization process aimed specifically at maintenance and based on a general qualitative analysis methodology. This process is rigorously defined in order to be repeatable and usable by people who are not acquainted with such analysis procedures. A basic feature of our approach is that maintenance changes are analyzed in order to understand the flaws in the change process. Guidelines are provided and a case study is shown that demonstrates the usefulness of the approach.

1 Introduction

As described in [HV92], numerous factors can affect software maintenance quality and productivity, e.g., the maintenance personnel experience profile and training, the way knowledge about the maintained systems is managed and conveyed to the maintainers and users, the maintenance organization, processes and standards in use, the initial quality of the software source code and its documentation. This last factor involves concepts such as self-descriptiveness, modularity, simplicity, consistency, expandability, and testability.

Because of the complexity of the phenomena studied, it is difficult for maintenance organizations to identify and assess the issues they have to address in order to improve the quality and productivity of their maintenance projects. Each project may encounter specific difficulties and situations that are not necessarily alike across all the organization's

maintenance projects. This may be due in part to variations in application domain, size, change frequency, and/or schedule/budget constraints. As a consequence, each project has first to be analyzed as a separate entity even if, later on, commonalities across projects may require similar solutions for improvement. Informally interviewing the people involved in the maintenance process would be unlikely to help determine accurately the real issues. Maintainers, users and owners would likely each give very different, and often contradictory, insights on the issues due to a somewhat incomplete and biased perspective.

Establishing a measurement program integrated into the maintenance process is likely to help any organization achieve an in-depth understanding of its specific maintenance issues and thereby lay a solid foundation for maintenance process improvement [RUV92]. However, defining and enacting a measurement program may take time and a short term, quickly operational substitute is needed in order to obtain a first quick insight, at low cost, into the issues to be addressed. Furthermore, defining efficient and useful measurement procedures first requires a characterization of the maintenance environment in which measurement takes place, i.e., organization structures, processes, issues, risks, etc. [BR88].

This paper presents a qualitative and inductive analysis methodology for performing objective project characterizations and thereby identifying their specific problems and needs. It is an implementation of the general qualitative analysis methodology defined in [SS92]. It encompasses a set of procedures which allows the determination of causal links between maintenance problems and flaws of the maintenance organization and process. Thus, a set of concrete steps for maintenance quality and productivity improvement can be taken based on a tangible understanding of the relevant maintenance issues. Moreover, this understanding provides a solid basis on which to define relevant software maintenance models and metrics.

Section 2 describes the phases, techniques and guidelines composing the methodology. Section 3 presents a case study of an orbit determination system maintained by the Flight Dynamics Division (FDD) of the NASA Goddard Space Flight Center for the last 26 years and still used daily for most operating satellites

This work was supported in part by NASA grant NSG-5123

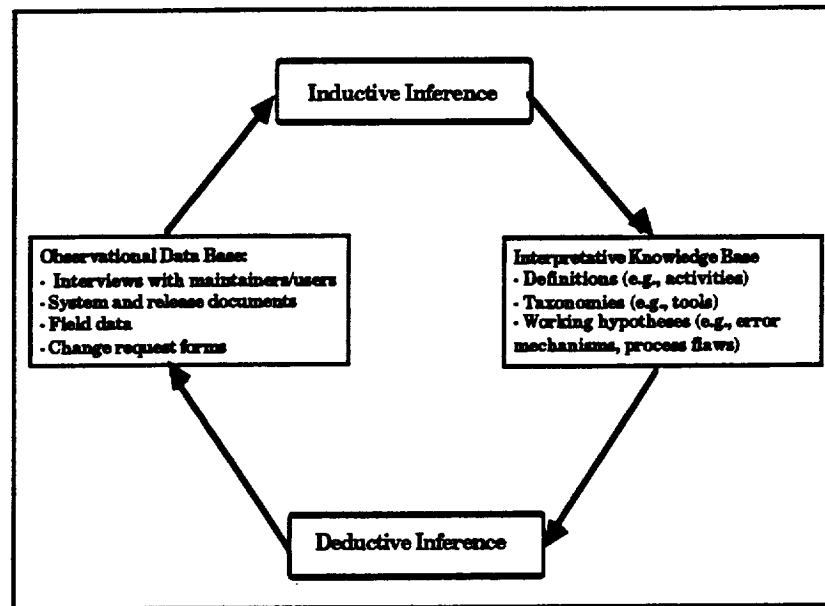


Figure 1: Qualitative Analysis Process for Software Maintenance

(GTDS: Goddard Trajectory Determination System). This study takes place in the framework of the NASA Software Engineering Laboratory (NASA-SEL), an organization aimed at improving FDD software development processes based on measurement and empirical analysis. Recently, responding to the growing cost of software maintenance, the SEL has initiated a program aimed at characterizing, evaluating and improving its maintenance processes. This paper is a first step in this direction. Section 4 outlines the main conclusions of the case study and the future research directions.

2 Causal Analysis of Maintenance Problems

In this section, we present a (mainly) qualitative methodology that allows for an in-depth characterization of maintenance projects at a relatively low cost. However, this approach could be easily augmented to integrate data collection and analysis and could thus provide more quantitative information (but at a higher cost).

2.1 A Qualitative Analysis Process

This characterization process is essentially an instantiation of the generic qualitative analysis process defined in [SS92]. Figure 1 illustrates at a high level our maintenance specific analysis process. It can be seen that it is a combination of both inductive and deductive inferences. Inductive inferences are based on the collected information, and deductive inferences occur when experimentally validating and refining our

taxonomies, process models, organizational models and working hypotheses. These deductive inferences then serve to refine the data collection process, which leads to refined and revised inductive inferences. The process continues in an iterative fashion.

We present below a general description of the process involved in preparing and performing characterizations of maintenance projects. Maintenance is defined here as any kind of enhancement, adaptation or correction performed on the software system once in operation. At the highest level of abstraction, parts of this process do not appear specific to maintenance and could also be used for development. However, the taxonomies and guidelines developed to support this process and presented in Section 2.2 are specifically aimed at maintenance.

Step 1 focuses on defining the organizational structures, i.e., organization entities, their communication channels and information flows. The process of producing a new release is then described and modeled in Step 2. It is important to note that we do not address here the issues related to emergency bug fixing procedures but only those relevant to regular product releases that go into configuration management. Step 3 maps generic activities into the release process in order to specify the type of work performed at each stage of the process. Then, a release (or several) has to be selected in order to define the set of changes on which the analysis will be performed (Step 4). In Step 5, relying on the work performed in Steps 1-3, information about the changes is collected and analyzed. Step 6 summarizes and abstracts from the results obtained in Step 5.

Although the steps are defined sequentially, they are really iterated within and across steps. As we learn more about the organization, we continue to refine the characterization models. The organizational and process models produced should include enough detail to allow Step 5 to be performed, but should not be so detailed as to obscure the maintenance process itself. We now define the steps in more detail:

1 Identify the organizational entities with which the maintenance team interacts and the organizational structure in which maintainers operate.

1.1 Identify distinct organizational entities, i.e., what are the distinct teams involved in the maintenance project? Usually, besides the maintainers themselves, the following entities are encountered: users, owners, QA team, configuration control team, change control board. However, their roles and prerogatives can differ significantly.

1.2 Characterize the working environment of each entity, i.e., support tools (see tool taxonomy in Section 2.2), internal organizational structure.

1.3 Characterize information flows between entities, i.e., what is the type (and amount when data available) of information, documentation, source code and other software artifacts flowing between organizational entities?

2 Identify the phases involved in the creation of a new system release.

2.1 Identify the phases as defined in the environment studied. At this stage, it is important not to map an *a priori* external/generic maintenance process model and vocabulary.

2.2 Each artifact (e.g., document, source code) which is input or output of each phase has to be determined and its content carefully described (see document taxonomy in Section 2.2).

2.3 The personnel in charge of producing and validating the output artifacts of each phase have to be identified and located in the organizational structure defined in Step 1.

3 Identify the generic activities involved in each phase.

3.1 Select (from the literature [C88, BC91]) or define a taxonomy of generic activities based on widely accepted definitions and used in the maintenance process. As a guideline, such a taxonomy is proposed in the next section.

3.2 Map these activities into each phase by reading the technical documents produced and interviewing the technical project leaders and maintainers about their real work habits. If possible,

collect effort data for each activity so that the importance of each activity in each phase can be assessed somewhat quantitatively.

4 Select one or several past releases for analysis.

We need to select releases on which we can analyze problems as they are occurring and thereby better understand process and organization flaws. However, because of time constraints, it is sometimes more practical to work on past releases. We present below a set of guidelines for selecting them:

- Recent releases are preferable since maintenance processes and organizational structure might have changed and this would make one's analysis somewhat irrelevant.

- Some releases may contain more complete documentation than others. Documentation has a very important role in detecting problems and cross-checking the information provided by the maintainers.

- The technical leader(s) of a release may have left the company whereas another release's technical leader may still be contacted. This is a crucial element since, as we will see, the causal analysis process will involve project technical leader(s) and, depending on his/her/their level of control and knowledge, possibly the maintainers themselves.

5 Analysis of the problems that occurred while performing the changes of the selected releases.

For each change (i.e., error correction, enhancement, adaptation) in the selected release(s), the following information should be acquired by interviewing the maintainers and/or technical leaders and by reading the related documentation (e.g., release documents):

I1. Determine the difficulty or error-proneness of the change.

I2. Determine whether the change difficulty could have been alleviated or the error(s) resulting from the change avoided and how?

I3. Evaluate the size of the change (e.g., # components, LOCs changed, added, removed).

I4. Assess discrepancies between initial & intermediate planning and actual effort / time.

I5. Determine the human flaw(s) (if any) that originated the error(s) or increased the difficulty related to the change. A taxonomy of human errors is proposed in Section 2.2.

I6. Determine the maintenance process flaws that led to the identified human errors (if any). A taxonomy of maintenance process flaws is proposed in Section 2.2.

I7. Try to quantify the wasted effort and/or delay generated by the maintenance process flaws (if any).

The knowledge and understanding acquired through steps 1-3 is necessary in order to understand, interpret and formalize the information of type I2, I5 or I6. As a guidance in conducting interviews, templates of questions will be provided in Section 2.2.

6 *Establish the frequency and consequences of problems due to flaws in the organizational structure and the maintenance process by analyzing the information gathered in Step 5.*

Based on these results, further complementary investigations (e.g., measurement based) related to specific issues that have not been fully resolved by the qualitative analysis process, should be identified. Moreover, a first set of suggestions for maintenance process improvement should be devised.

For those steps which are iterative, we map the appropriate step back into the qualitative analysis process (Figure 1). Thereby, we show how our characterization process fits into the more general qualitative analysis methodology presented above. In this context, a step usually corresponds to a set of iterations of the qualitative analysis process. Thus for each step we have the *input* to that step which defines the *Observational Database* (ODB), the *output* of each step which contains the resulting characterization models that go into the *Interpretative Knowledge Base* (IKB), and a validation procedure which helps verify that the characterization models are correct. The pieces of information which compose the ODB are given in decreasing order of importance at each step. The order and content of the ODB varies at each step since the analysis focus is progressively shifting [SS92].

Step 1: Model organizational structures

Input: maintenance standards definition document, interviews, sample of release documents, organization chart

Output: organizational model (roles, agents, teams, information flow, etc.)

Validation:

- . Are all the standard documents and artifacts included in the modeled information flow?
- . Do we know who produces, validates, and certifies the standard documents and artifacts?
- . Are all the people referenced in the release documents a part of the organization scheme?

Steps 2, 3: Model process and map activities into process phases

Input: maintenance standards definition document, interviews, release documents

Output: process model

Validation:

. Are all the people in the process model a part of the organization scheme?

. Do the documents and artifacts included in the process model match those of the information flow of the organization model?

. Is the mapping between activities and phases complete, i.e., exhaustive set of activities, complete mapping?

. Are the taxonomies of maintenance tools, methods, and activities adequate, i.e., unambiguous, disjoint and exhaustive classes?

Step 5: Perform causal analysis

Input: interviews, change request forms, release documents, organization model, process model, maintenance standards definition document

Output: causal analysis

Validation:

. Are the taxonomies of errors and maintenance process flaws adequate, i.e., unambiguous, disjoint and exhaustive classes? This is checked against actual change data and validated during interviews with maintainers.

2.2 Guidelines and Taxonomies

This section presents a set of guidelines aimed at facilitating the characterization process described in the previous section. These guidelines are mainly composed of taxonomies distinguishing maintenance activities, errors and maintenance process flaws. In addition, a set of questions which can be used during maintainers' interviews and for each change is provided.

Step 1: Identify organizational entities

Taxonomies of Maintenance Tools and Methods (Step 1.2)

The maintenance tools and methods available to maintainers can be used to understand the maintenance process, and identify potential sources of problems. The following paragraphs represent the first level of abstraction of environment characteristics' taxonomies that should be used to characterize the change framework:

- Maintenance Tools: Impact analysis & planning tool ; Tools for automated extraction & representation of control and data flows ; Debugger ; Cross-referencer ; Regression testing environment (data generation, execution, and analysis of results) ; Information system linking documentation and code.
- Maintenance Methods are characterized by the following taxonomy: rigorous impact analysis, planning, and scheduling procedures ; Systematic and disciplined update procedures of the user and system

documentation ; Communication channels and procedures with the users ;

A Taxonomy of Maintenance Documentation (Step 1.3)

The type of documentation related to a software system, which may be available to maintainers, can be defined by a generic taxonomy as shown below. Documentation has been described as one of the most important factors affecting the maintainability of a software system [HA93, P94]. This is why it is important to define precisely what should be contained in a complete set of documentation (either on-line or off-line) for maintenance. Such a taxonomy can be used as a guideline to define the maintenance organization. Also, when some of these documents appear to be missing, potential sources of maintenance problems may be identified. Based on the literature [BC91] on the subject and our own experience, we propose the following taxonomy:

- Product-related: Software requirements specifications ; Software design specifications ; Software product specifications
- Process-related: Test plans ; Configuration management plan ; Quality assurance plan ; Software development plan
- Support-related: Software user's manual ; Computer systems operator's manual ; Software maintenance manual ; Firmware support manual

Step 3: Identify the generic activities involved in each phase.

Generic Description of Maintenance Activities (Step 3.1)

Acronym	Activity
DET	Determination of the need for a change
SUB	Submission of change request
UND	Understanding requirements of changes: localization, change design prototype
IA	Impact analysis
CBA	Cost/benefit analysis
AR	Approval/Rejection/priority assignment of change request
SC	Scheduling of task
CD	Change design
CC	Code changes
UT	Unit testing of modified parts i.e., has the change been implemented?
IT	integration testing,

i.e., does the changed part interface correctly with the system?

RT	Regression testing, i.e., does the change have any unwanted side effects?
AT	Acceptance testing i.e., does the new release fulfill the system requirements?
USD	Update system & user documentation
SA	Standards characterizations; quality assurance procedures
IS	Installation
PIR	Post-installation review of changes
EDU	Education/training regarding the application domain/system

All these activities usually contain an overhead of communication (meeting + release document writing) with owner/users, management hierarchy and other maintainers which should be estimated. This is possible, through data collection or by interviewing maintainers (e.g., Delphi method).

Step 5: Perform causal analysis

Questions asked for each change in selected release(s) (Items 11-14)

The following list describes a set of questions for which answers can be provided by maintainers and/or release standard documents. These questions attempt to capture the information necessary for the identification of maintenance process flaws.

1 - Description of the change

1.1 Localization

- . subsystem(s) affected
- . module(s) affected
- . inputs/outputs affected

1.2 Size

- . LOCs deleted, changed, added
- . Modules examined, deleted, changed, added

1.3 Type of change

- . Preventive changes: improvement of clarity, maintainability or documentation.
- . Enhancement changes: add functionalities, optimization of space/time/accuracy
- . Adaptive changes: adapt system to change of hardware and/or platform
- . Corrective changes: corrections of development errors.

2 - Description of the change process

2.1 effort, elapsed time

2.2 maintainer's expertise and experience

- . How long has the person been working on the system
- . How long has the person been working in this application domain?

2.3 Did the change generate a change in any document? Which document(s)?

3 - Description of the problem

3.1 Were some errors committed?

- . Description of the errors (see taxonomies below)
- . Perceived cause of the errors: maintenance process flaw(s)

3.2 Difficulty

- . What made the change difficult?
- . What was the most difficult activity associated with the change?

3.3 How much effort was wasted (if any) as a result of maintenance process flaws?

3.4 What could have been done to avoid some of the difficulty, errors (if any)?

Taxonomies of human errors (Item 15)

Note that we are exclusively referring to errors occurring during the maintenance process, not errors resulting from the development.

• Error Origin: when did the misunderstanding occur?

- . Change requirements analysis
- . Change localization analysis
- . Change design analysis
- . Coding

• Error domain: what caused it?

- . Lack of application domain knowledge: *operational constraints (user interface, performance), mathematical model*
- . Lack of system design or implementation knowledge: *data structure or process dependencies, performance or memory constraints, module interface inconsistency*
- . Ambiguous or incomplete requirements
- . Language misunderstanding <semantic, syntax>
- . Schedule pressure
- . Existing uncovered fault
- . Oversight.

Determining the origin and cause of the errors will help determine their possible causal relationships to maintenance process flaws in the taxonomy presented below.

Taxonomy of Maintenance Process flaws (Item 16)

• Organizational flaws:

- . communication: Interface problems, information flow "bottlenecks" in the communication between the maintainers and the
- . users
- . management hierarchy
- . quality assurance (QA) team
- . configuration management team
- . roles:
- . prerogatives and responsibilities are not fully defined or explicit
- . incompatible responsibilities, e.g., development and QA
- . process conformance: no effective structure for enforcing standards and processes

• Maintenance methodological flaws

- . Inadequate change selection & priority assignment process
- . Inaccurate methodology for planning of effort, schedule, personnel
- . Inaccurate methodology for impact analysis
- . Incomplete, ambiguous protocols for transfer, preservation and maintenance of system knowledge
- . Incomplete, ambiguous definitions of change requirements
- . Lack of rigor in configuration (versions, variations) management and control
- . Undefined / unclear regression testing success criteria.

• Resource shortages

- . Lack of financial resources allocated, e.g., necessary for preventive maintenance, unexpected problems unforeseen during impact analysis.
- . Lack of tools providing technical support (see previous tool taxonomy)
- . Lack of tools providing management support (i.e., impact analysis, planning)

• Low quality product(s)

- . Loosely defined system requirements
- . Poor quality design, code of maintained system
- . Poor quality system documentation
- . Poor quality user documentation

- Personnel-related issues
 - . Lack of experience and/or training with respect to the application domain
 - . Lack of experience and/or training with respect to the system requirements (hardware, performance) and design
 - . Lack of experience and/or training with respect to the users' operational needs and constraints

In order to demonstrate the feasibility and usefulness of the above approach, we present the following case study.

3 A Case Study

This case study is intended to provide actual examples and results of the change causal analysis process described in previous sections. We first present the maintained system used as a case study. Then, the specific maintenance organization and process are described in detail according to the template provided in Section 2.1. Examples of change causal analyses are shown and the lessons learned resulting from this analysis process are presented.

3.1 System History and Description

GTDS is a 26 year old, 250 KLOC, FORTRAN orbit determination system. It is public domain software and, as a consequence, has a very large group of users all over the world. Usually, 1 or 2 releases are produced every year in addition to mission specific versions that do not go into configuration management right away (but are integrated later on to a new version by going through the standard release process). Like most maintained software systems, very few of the original developers are still present in the organization, but the turnover of the maintenance team is low compared to other maintenance organizations. However, turnover still remains a crucial issue in this environment.

3.2 Modeling of the Maintenance Organization and Processes

During the process of building a new release of GTDS, different organizational entities interact in different ways. By performing Step 1 of the characterization process described in Section 2.1, two types of entities and five types of interactions (i.e., differentiated according to the purpose of the information flow) were identified.

The entities, teams and groups, are represented in Figure 2 by boxes and ellipses, respectively. Teams

are persistent organizational structures; groups are composed of members of several different teams, and are dynamic entities in the sense that they only exist when group members meet. These groups have been designed to facilitate communication between teams and decision making.

In the five interaction types identified, information was used for the following purposes: decision - decision based on information provided; review - review of documents; approval - approval of documents or plans; transformation - supplied information product is transformed into another information product; and information - dissemination of information.

Teams:

- . *Testers*: they present acceptance test plans, perform acceptance test and provide change requests to the maintainers when necessary.

- . *Owners / Users*: they suggest, control and approve performed changes.

- . *Product Assurance Organization (PAO)*: They control maintainers' work, e.g., conformance to standards, attend release meetings, audit delivery packages. They have a different management from the maintenance team.

- . *Configuration Management (FDCM)*: They integrate updates into the system. Coordinate the production and release of versions of the system. Provide tracking of change requests.

- . *Maintenance management*: They grant preliminary approvals of maintenance change requests and release definitions.

- . *Maintainers*: They analyze changes, make recommendations, perform changes, perform unit and change validation testing after linking the modified units to the existing system, perform validation and regression testing after they get back the recompiled system from the *FDCM* team.

Groups:

- . *Software Management Planning Board (SMPB)*: Their main goal is to address management issues that run across maintenance projects. For example, they help resolve conflicts between owners and maintainers and review release planning documents. Also, they allow task leaders and higher level managers to exchange relevant information about the evolution of their respective systems. However, *SMPB* has no official function. The board is composed of the task leader, section manager, department manager, and operation manager.

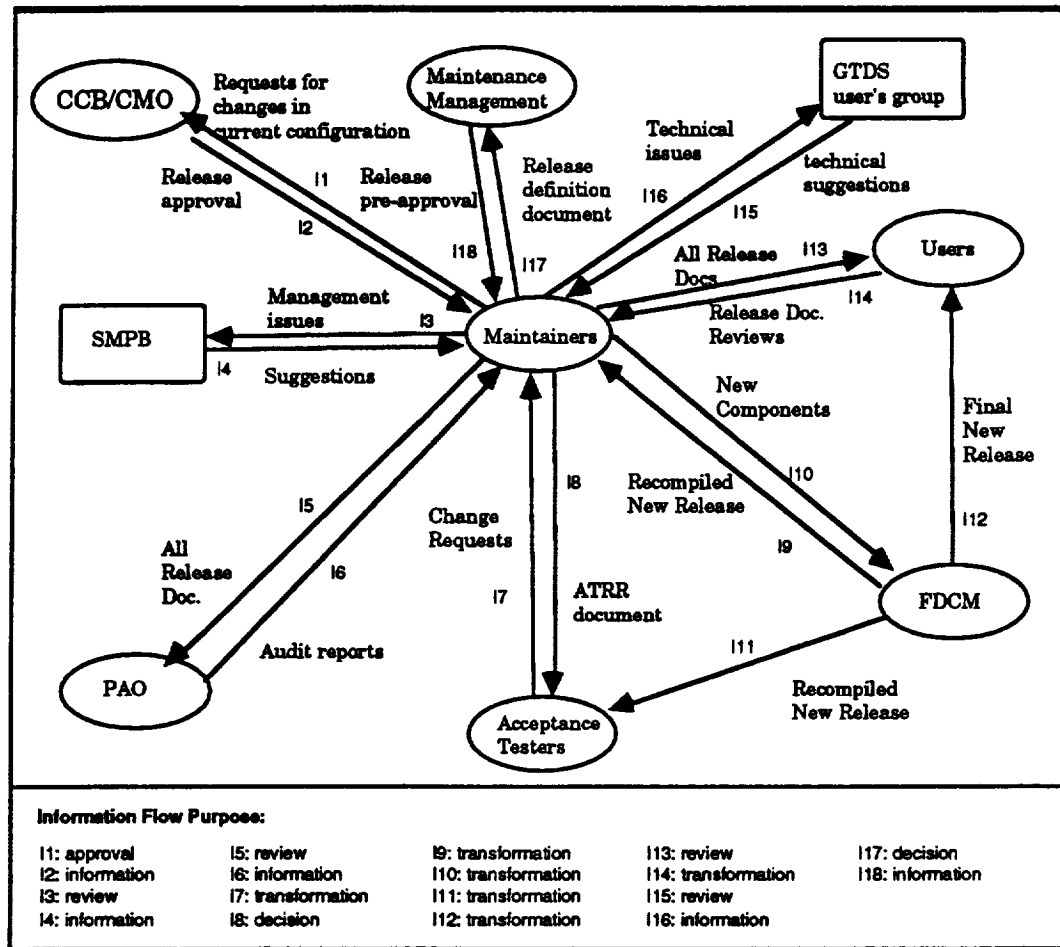


Figure 2: Information Flow within the Maintenance Organization

Configuration Control Board and Configuration Management Office (CCB/CMO): They are officially responsible for all changes to configured software and the allocated budget. Their goal is to ensure that the production of new releases is consistent with the long-term goals of the organization. It is composed of high-level managers.

GTDS user's group: It is a forum for discussion of technical issues but has no official function. It is composed of users, maintainers, and testers.

The process described below represents our understanding of the working process for a release of GTDS and the mapping into standard generic activities. This combines the information gained from Steps 2 and 3 of the characterization process. Phases, their associated inputs/outputs and activities are presented below. Activity acronyms are used as defined in Section 2.2. In this case, each phase milestone in a release is

represented by the discussion, approval and distribution of a specific release document.

1. Change analysis

Input: change requests from software owner + priority list

Output: Release Content Review (RCR) document which contains change design analysis, prototyping, and cost/benefit analysis that may result in a priority change to be discussed with the software owner/user.

Activities: UNDR, IA, CBA, CD, some CC, UT and IT (for prototyping)

2. RCR meeting

Input: Release Content Review document proposed by maintainers is discussed, i.e., change priority, content of release.

Output: Updated Release Content Review document

Activities: AR, SA (QA engineers are reviewing the release documents and attending the meeting)

3. Solution analysis

Input: Updated Release Content Review document

Output: devise technical solutions based on prototyping analysis they performed in Step 1, Release Design Review (RDR)

Activities: SC, CD, CC, UT, (preparation of test strategy for) IT (based mainly on equivalence partitioning)

4. RDR meeting

Input: RDR documentation

Output: approved (and possibly modified) RDR documentation

Activities: review and discuss CC, UT, (plan for) IT, SA

5. Change implementation and test

Input: RDR + prototype solutions (phases 1, 3)

Output: changes are completed ; change validation test is performed with new compiled components linked to unchanged components of the current system version ; regression testing is performed on the system recompiled from scratch (provided by the FDCM team) ; a report with the purpose of demonstrating that the system is ready for acceptance test is produced: Acceptance test readiness review document (ATRR)

Activities: IT, RT, USD

6. ATRR meeting

Input: Acceptance test readiness review document

Output: The changes are discussed and validated and the used testing strategy is discussed. The acceptance test team presents its acceptance testing plan.

Activities: review the current output of IT, SA

7. Acceptance test

Input: the new GTDS release and all release documentation

Outputs: A list of Software Change requests (SCRs) is provided to the maintainers. These changes correspond to inconsistencies between the new release and the general system requirements.

Activities: AT

Step 1, 2, and 3 required several iterations before there was sufficient validation of the resulting characterization of the organization, phases and documents. As part of Step 2, for each of the standard documents generated during the releases of GTDS studied, we determine who produces it, who approves it, and what additional relevant information and data they contain. When doing so, we have to look for possible inconsistencies between the organization model (Step 1) and the identified producers/approvers of the documents.

• Document 1: Release Content Review (RCR):

Producer: maintenance team

Approvers: users, maintenance management, CCB

Content:

- . change requirement description
- . description of error (if any) that originated the change
- . design of a prototype solution
- . schedule, effort plans
- . impact analysis assessment

• Document 2: Release Design Review (RDR):

Producer: maintenance team

Approvers: users, CCB

Content:

- . identification of modified units
- . a definitive solution is proposed
- . rough cost/schedule estimates
- . testing guidelines: mainly equivalence partitioning classes
- . definition of the test success criterion

• Document 3: Acceptance Test Readiness Review (ATRR):

Producers: maintenance team, acceptance test team (test plan)

Approvers: CCB, testers

Content:

- . results of test cases and benchmarks (regression testing)
- . screen printouts, short reports
- . Acceptance test plan

• Document 4: Delivery package:

Producer: maintenance team

Approvers: CCB

Content:

- . cause of error (if any)
- . effort breakdown: analysis, design, code, test
- . # components examined, modified, added, deleted.
- . # Locs modified, added, deleted

As specified in Step 4 of our process, we selected a release for analysis. This release was quite recent, most of the documentation identified in Step 2 was available, and most importantly, the technical leader of the release was available for additional insights and information.

Step 5 involved a causal analysis of the problems observed during maintenance and acceptance test of the releases studied. These problems were linked back to a precise set of issues belonging to taxonomies presented in Section 2.2. Figure 3 summarizes Step 5 as instantiated for this case study.

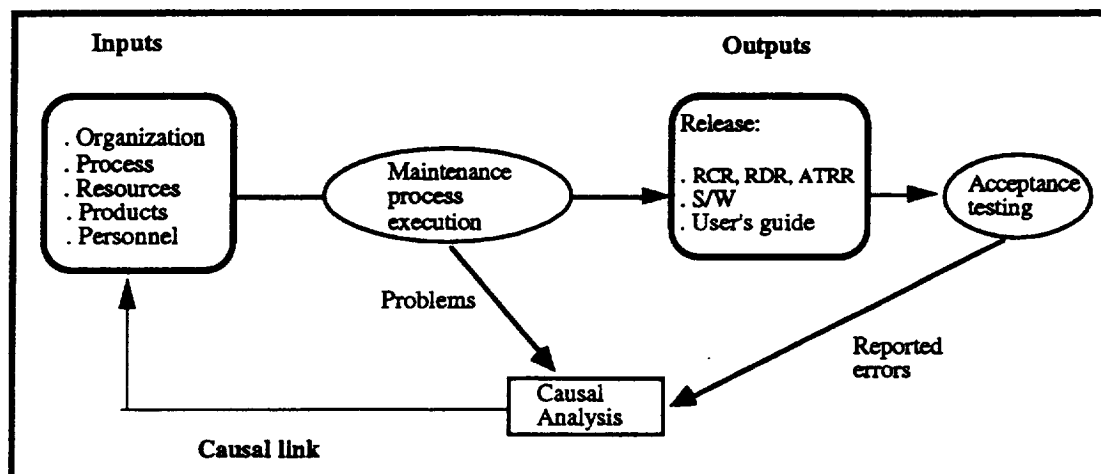


Figure 3: Causal Analysis in GTDS

In order to illustrate Step 5, we provide below an example of causal analysis for *one* of the changes in the selected release. Implementation of this change resulted in 11 errors that were found by the acceptance test team, 8 of which had to be corrected before final delivery could be made. In addition, a substantial amount of rework was necessary. Typically, changes do not generate so many subsequent errors, but the flaws that were present in this change are representative of maintenance problems in GTDS. In the following paragraphs, we discuss only *two* of the errors generated by the change studied.

- Increased difficulty related to change (rework)

Description: Initially, users requested an enhancement to existing GTDS capabilities (change 642). The enhancement involved vector computations performed over a given timespan. This enhancement was considered quite significant by the maintainers, but users failed to supply adequate requirements and did not attend the RCR meeting. Users did not report their dissatisfaction with the design until ATRR meeting time, at which time requirements were rewritten and maintainers had to perform rework on their implementation. This change took a total of 3 months to implement, of which at least 1 month was attributed to several flaws in the process.

Maintenance process flaw(s): organizational: a lack of clear definitions of the prerogatives/duties of users with respect to release document reviews and meetings (roles), and a lack of enforcement of the release procedure (process conformance); maintenance methodological flaw: incomplete, ambiguous definitions of change requirements.

- Errors caused by change 642

The implementation of the change itself resulted in an error (A1044) found at the acceptance test phase. When the correction to A1044 was tested, an error (A1062) was found that could be traced back to both 642 and A1044.

A1044

Description: Vector computations at the endpoints of the timespan were not handled correctly. But in the requirements it was not clear whether the endpoints should be considered when implementing the solution.

Error origin: change requirement analysis

Error domain: ambiguous and incomplete requirements

Maintenance process flaw(s): organizational: communication between users and maintainers, due in part to a lack of defined standards for writing change requirements; maintenance methodological flaw: incomplete, ambiguous definitions of change requirements.

A1062

Description: One of the system modules in which the enhancement change was implemented has two processing modes for data. These two modes are listed in the user manual. When run in one of the two possible processing modes, the enhancement generated a set of errors, which were put under the heading A1062. At the phase these errors were found, the enhancement had already successfully passed the tests for the other processing mode. The maintainer should have designed a solution to handle both modes correctly.

Error origin: change design analysis.

Error domain: lack of application domain knowledge.

Maintenance process flaw(s): personnel-related: lack of experience and/or training with respect to the application domain.

The next section presents in detail the results of performing Step 6.

3.3 Lessons Learned about the Studied Maintenance Project

The lessons learned are classified according to the taxonomy of maintenance flaws defined in Section 2.2. By performing an overall analysis of the change causal analysis results (Step 6), we abstracted a set of issues classified as follows:

Organization

- There is a large communication cost overhead between maintainers and users, e.g., release standard documentation, meetings, management forms. In an effort to improve the communication between all the participants of the maintenance process, non-technical, communication-oriented activities have been emphasized. At first glance, this seems to represent about 40% (rough approximation) of the maintenance effort. This figure seems excessive, especially when considering the apparent communication problems (next paragraph).

- Despite the number of release meetings and documents, disagreements and misunderstandings seem to disturb the maintenance process until late in the release cycle. For example, design issues that should be settled at the end of the RDR meeting keep emerging until acceptance testing is completed.

As a result, it seems that the administrative process and organization scheme should be investigated in order to optimize communication and sign-off procedures, especially between users and maintainers.

Process

- The tools and methodologies used have been developed by maintainers themselves and do not belong to a standard package provided by the organization. Some ad hoc technology transfer seems to take place in order to compensate for the lack of a global, commonly agreed upon strategy.

- The task leader has been involved in the maintenance of GTDS for a number of years. His expertise seems to compensate for the lack of system documentation. He is also in charge of the training of new personnel (some of the easy changes are used as an opportunity for training). Thus, the process relies heavily on the expertise of one or two persons.

- The fact that no historical database of changes exists makes some changes very difficult. Maintainers very often do not understand the semantics of a piece of

code added in a previous correction. This seems to be partly due to emergency patching (during a mission) which was not controlled and cleaned up afterwards (this has recently been addressed), a high turnover of personnel and a lack of written requirements with respect to performance, precision and platform configuration constraints.

- For many of the complex changes, requirements are often ambiguous and incomplete, from a maintainer's perspective. As a consequence, requirements are often unstable until very late in the release process. While prototyping might be necessary for some of them, it is not recognized as such by the users and maintainers. Moreover, there is no well defined standard for expressing change requirements in a way suitable to both maintainers and users.

Products

- System documentation (besides the user's guide) is not fully maintained and not trusted by maintainers. Source code is currently the only reliable source of information used by maintainers.

- GTDS has a large number of users. As a consequence, the requirements of this system are complex with respect to the hardware configurations on which the system must be able to run, the performance and precision needs, etc. However, no requirement analysis document is available and maintained in order to help the maintainers devise optimal change solutions.

- Because of budget constraints, there is no document reliably defining the hardware and precision requirements of the system. Considering the large number of users and platforms on which the system runs, and the rapid evolution of users' needs, this would appear necessary in order to avoid confusion while implementing changes.

People

- There is a lack of understanding of operational needs and constraints by maintainers. Release meetings were supposed to address such issues but they seem to be inadequate in their current form.

- Users are mainly driven by short term objectives which are aimed at satisfying particular mission requirements. As a consequence, there is a very limited long term strategy and budget for preventive maintenance. Moreover, the long term evolution of the system is not driven by a well defined strategy and maintenance priorities are not clearly identified.

As a general set of recommendations and based on the analysis presented in this paper, we suggest the following set of actions:

- A standard (that may simply contain guidelines and checklists) should be set up for change

requirements. Both users and maintainers should give their input with respect to the content of this standard.

- The conformance to the defined release process should be improved, e.g., through team building, training. In other words, the release documents and meetings should more effectively play their specified role in the process, e.g., the RDR meeting should settle all design disagreements and inconsistencies.

- The parts of the system that are often changed and highly convoluted (as a result of numerous modifications) should be redesigned and documented for more productive and reliable maintenance. Technical task leaders should be able to point out the sensitive system units.

4 Conclusion

Characterizing and understanding software maintenance processes and organizations are necessary, if effective management decisions are to be made and if adequate resource allocation is to be provided. Also, in order to plan and efficiently organize a measurement program—a necessary step towards process improvement [BR88]—, we need to better characterize the maintenance environment and its related issues. The difficulty of performing such a characterization stems from the fact that the people involved in the maintenance process, who have the necessary information and knowledge, cannot perform it because of their inherently biased perspective on the issues. Therefore, a well defined characterization process, which is cost-effective, objective, and applicable by outsiders, needs to be devised.

In this paper, we have presented such an empirically refined characterization process which has allowed us to gain an in-depth understanding of the maintenance issues involved in a particular project, the GTDS project. We have been able to gather objective information on which we can base management and technical decisions about the maintenance process and organization. Moreover, this process is general enough to be followed in most of the maintenance organizations.

However, such a qualitative analysis is a priori limited since it does not allow us to quantify precisely the impact of various organizational, technical, and process related factors on maintenance cost and quality. Thus, the planning of the release is sometimes arbitrary, monitoring its progress is extremely difficult, and its evaluation remains subjective.

Hence, there is a need for a data collection program for GTDS and across all the maintenance projects of our organization. In order to reach such an objective, we will base the design of such a measurement program on the results provided by this study. In addition, we need to model more rigorously the maintenance organization and processes so that precise evaluation criteria can be defined [SB94].

This approach will be used to analyze several other maintenance projects in the NASA-SEL in order to better understand project similarities and differences in this environment. Thus, we will be able to build better models of the various classes of maintenance projects.

Acknowledgments

We are grateful to Steve Condon, Walcelio Melo, Carolyn Seaman, Barbara Swain and Jon Valett for reviewing early drafts of this paper. We also would like to thank Amy Bleich for helping us to analyze the release documents.

References

- [BC91] K. Bennett, B. Cornelius, M. Munro, D. Robson, "Software Maintenance", *Software Engineering Reference Book*, Chapter 20, Butterworth-Heinemann Ltd, 1991
- [BR88] V. Basili and H. Rombach, "The TAME Project: Towards Improvement-Oriented Software Environments", *IEEE Trans. Software Eng.*, 14 (6), June, 1988.
- [C88] N. Chapin, "The Software Maintenance Life-Cycle", CSM'88, Phoenix, Arizona, 1988.
- [HA93] C. Hartzman, C. Austin, "Maintenance Productivity: Observations Based on an Experience in a Large System Environment", CASCON'93, Toronto, Canada, 1993
- [HV92] M. Hariza, J.F. Voidrot, E. Minor, L. Pofelski, and S. Blazy, "Software Maintenance: An analysis of Industrial Needs and Constraints", CSM'92, Orlando, Florida.
- [P94] D. Parnas, "Software Aging", ICSE 16th, May 1994, Sorrento, Italy.
- [RUV92] D. Rombach, B. Ulery and J. Valett, "Toward Full Cycle Control: Adding Maintenance Measurement to the SEL", *Journal of systems and software*, May 1992.
- [SB94] C. Seaman, V. Basili, "OPT: An Approach to Organizational and Process Improvement", AAAI 1994 Spring Symposium Series, Stanford University, March 1994.
- [SS92] A. Shelly, E. Sibert, "Qualitative Analysis: A Cyclical Process Assisted by Computer", *Qualitative Analysis*, pp 71-114, Oldenbourg Verlag, Munich, Vienna, 1992

Defining and Validating High-Level Design Metrics¹

Lionel Briand*, Sandro Morasca**, Victor R. Basili*

* Computer Science Department
University of Maryland, College Park, MD, 20742
{lionel, basili}@cs.umd.edu

** Dipartimento di Elettronica e Informazione
Politecnico di Milano
Piazza Leonardo da Vinci 32, I-20133 Milano, Italy
morasca@elet.polimi.it

Abstract

The availability of significant metrics in the early phases of the software development process allows for a better management of the later phases, and a more effective quality assessment when software quality can still be easily affected by preventive or corrective actions. In this paper, we introduce and compare four strategies for defining high-level design metrics. They are based on different sets of assumptions (about the design process) related to a well defined experimental goal they help reach: identify error-prone software parts. In particular, we define ratio-scale metrics for cohesion and coupling that show interesting properties. An in-depth experimental validation, conducted on large scale projects demonstrates the usefulness of the metrics we define.

1 Introduction

Software metrics can help address the most critical issues in software development and provide support for planning, predicting, monitoring, controlling, and evaluating the quality of both software products and processes [BR88, F91]. Most existing software metrics attempt to capture characteristics of software code [F91]; however, software code is just *one* of the artifacts produced during software development, and, moreover, it is only available at a late stage. It is widely recognized that the production of better software requires the improvement of the early development phases and the artifacts they produce:

¹ This work was supported in part by NASA grant NSG-5123, UMIACS, and NSF grant 01-5-24845. Sandro Morasca was also supported by grants from MURST and CNR.

The production of better specifications and better designs reduces the need for extensive review, modification, and rewriting not only of code, but of specifications and designs as well. As a result, this allows the software organization to save time, cut production costs, and raise the final product's quality.

Early availability of metrics is a key factor to a successful management of software development, since it allows for

- early detection of problems in the artifacts produced in the initial phases of the life-cycle (specification and design documents) and, therefore, reduction of the cost of change—late identification and correction of problems are much more costly than early ones;
- better software quality monitoring from the early phases of the life-cycle;
- quantitative comparison of techniques and empirical refinement of the processes to which they are applied;
- more accurate planning of resource allocation, based upon the predicted error-proneness of the system and its constituent parts.

In this paper, we will focus on high-level design metrics for software systems. A number of studies have been published on software design metrics in recent years. It has been shown that system architecture has an impact on maintainability and error-proneness [HK84, G86, R87, R90, S90, SB91, Z91, AE92, BTH93, BBH93]. These studies have attempted to capture the design characteristics affecting the ease of maintaining and debugging a software system. Most of the design metrics are based on information flow between subroutines or declaration counts. We think that, even though it provides an interesting insight into the program structure, this should not be the only strategy to be investigated, since many other types of program features and relationships are *a priori* worth studying. Moreover, there is a need for comparison between strategies in order to identify worthwhile research directions and build accurate prediction models.

Besides this focus on information flow, most of the existing approaches share two common characteristics. (1) They define metrics without making clear assumptions about the contexts (i.e., processes, problem domain, environmental factors, etc.) in which they can be applied (with the exception of [AE92], where this issue was partially addressed). This implies they should have general validity, and be applicable to different environments and problem domains. (2) There are not fully explicit goals, for whose achievement the metrics are defined. This may cause problems in their application, since they may be defined based on implicit assumptions which the context may not satisfy; interpretation,

since their meaning is not clear; and validation [IS88, K88], since their relevance with respect to a clearly stated goal is not established.

The definition of universal metrics (like in physical sciences) is an acceptable long-term goal, which, however, is only achievable after we gain better insights into specific processes from specific perspectives in the short term. It is our opinion that the definition of a metric should be driven by both the characteristics of the context or family of contexts in which it is used, and one or more clearly stated goals that it helps reach. In other words, the assumptions underlying the defined metrics should rely on a deep knowledge of the context and should be precisely related to a stated goal. After this, the defined metrics must undergo a thorough experimental validation, to assess their significance and usefulness with respect to the stated goals. Last, based on the experimental evidence, metrics may be refined and modified, to better achieve the goals and comply with the process characteristics.

The goal of the research documented in this paper is to define and validate a set of high-level design metrics to evaluate the quality of the high-level design of a software system with respect to its error-proneness, understand what high-level design characteristics are likely to make software error-prone, and predict the error-proneness of the code produced.

We introduce four families of metrics, which are based on different types of mathematical abstractions of program designs [MGBB90]. In particular, we introduce a family of metrics based on data declaration dependency links (Section 2.2.4). This strategy allows us to introduce metrics for cohesion (Section 2.2.4.1) and coupling (Section 2.2.4.2) [F91] that are characterized by interesting properties and are based on consistent principles. Such a consistency is important because it should facilitate future research on quantitative tradeoff mechanisms between coupling and cohesion, i.e., variations can be expressed using consistent measurement units. Other metric families include: metrics based on declaration counts (Section 2.2.1), metrics based on the USES relationships between modules [GJM92] (Section 2.2.2), and metrics based on the IS_COMPONENT_OF relationships [GJM92] (Section 2.2.3).

In addition, we experimentally compare and validate the metrics introduced in Section 2 on three NASA projects. The results are shown in Section 3. In Section 4, we summarize the lessons we have learned, and outline directions for future research activities based on these lessons.

2 Defining Metrics for High-level Design

In this section, we first introduce the basic concepts of high-level design and the terminology we will use in the paper (Section 2.1). We then define, based on the goals stated in Section 1 and context assumptions, four families of high-level design metrics (Section 2.2).

2.1 Basic Definitions

Our object of study is the high-level design of a software system. To define it, we will start from its elementary constituents: software modules.

In the literature, there are two commonly accepted definitions of modules. The first one sees a module as a routine, either procedural or functional, and has been used in most of the design measurement publications [M77, CY79, HK84, R87, S90]. The second definition, which takes an object-oriented perspective, sees a module as a collection of type, data, and subroutine definitions, i.e., a provider of computational services [BO87, GJM92]. In this view, a module is the implementation of an Abstract Data Type / Object. In this paper, unless otherwise specified, we will use the term subroutine for the first category, and reserve the term module for the second category. Modules are composed of two parts: interface and body (which may be empty). The interface contains the computational resources that the module makes visible for use to other modules. The body contains the implementation details that are not to be exported.

At a higher level of abstraction, modules can be seen as the components of higher level aggregations, as defined below.

Definition 1: Library Module Hierarchy (LMH).

A library module hierarchy is a hierarchy where nodes are modules and subroutines, arcs between modules are IS_COMPONENT_OF [GJM92] relationships, and there is just one top level node, which is a module.

◇

In the remainder of this paper, we will define concepts and metrics that can be applied to both modules and LMHs, which are the most significant syntactic aggregation levels below the subsystem level. For short, we will use the term *software part (sp)* to denote either a module or an LMH.

In the high-level design phase of a software system, only module and subroutine interfaces and their relationships are defined—module body and subroutine detail design is

carried out at low-level design time. Therefore, we define the high-level design of a software system as follows.

Definition 2: High-level Design

The high-level design of a software system is a collection of module and subroutine interfaces related to each other by means of USES [GJM92] and IS_COMPONENT_OF relationships. No body information is yet available at this stage.

◊

2.2 Strategies to Define High-level Design Metrics

In this section, we investigate several strategies for defining high-level design metrics. This appears necessary at this stage of knowledge, where we can only rely on very limited theoretical and empirical ground to help us identify interesting concepts, relationships and objects of study. One of the results of this investigation is to provide directions to focus our research on a smaller set of strategies and concepts.

Some of the concepts introduced in this section cannot be directly mapped onto all imperative languages, because not all of them allow the implementation of Abstract Data Types/Objects. However, these concepts are shared by many modern programming languages.

As we said in the Introduction, context assumptions are necessary to define metrics that are applicable and useful. Therefore, we list a context assumption for each of the metrics of the four strategies we introduce below. We do not assume that all of these process assumptions are equally important, i.e., not all of the process characteristics we take into account have an equal impact on software error-proneness.

2.2.1 Declaration Counts

These metrics are counts of data declarations, associated with a software part, that are imported, exported or declared locally.

Metric 1: Local.

Local(sp) will denote the number of locally defined data declarations of a software part *sp*.

Assumption A-LO.

The count of declarations of a software part may be seen as a measure of size, which is known to be associated with errors, i.e, the larger the set of declarations, the more likely the errors.

◇

Metric 2: Global.

Global(sp) will denote the number of external data declarations visible from a software part *sp*.

Assumption A-GL.

The larger the number of external declarations visible in a software part, the larger the number of external concepts to be understood and used consistently, the higher the likelihood of error.

◇

Metric 3: Scope.

Scope(sp) will denote the number of external data declarations for which the data declarations of a software part *sp* are visible.

Assumption A-SC.

The larger the number of data declarations in the scope of the software part, the larger the number of contexts of use, the more likely it is to be inadequate to fulfill the needs of the declarations in the scope.

◇

2.2.2 Metrics Based on the USES Relationships

These metrics capture the dependencies between software parts based on the USES relationships of the system.

Metric 4: Imported Software Parts.

ISP(sp) will denote the number of software parts imported and used by a software part *sp*.

Assumption A-ISP.

The larger the number of used external software parts, the larger the context to be understood, the more likely the occurrence of an error.

◇

Metric 5: Exported Software Parts.

$ESP(sp)$ will denote the number of software parts that use a software part sp .

Assumption A-ESP.

The larger the number of contexts of use of a software part, the larger the number of services it provides, the more flexible it must be, and, as a consequence, the more likely the occurrence of error.

◇

2.2.3 Metrics Based on the IS_COMPONENT_OF Relationships

These metrics capture information about the structure of the IS_COMPONENT_OF graph.

Metric 6: Maximum/Average Depth.

$Max_Depth(sp) / Avg_Depth(sp)$ will denote the maximum/average depth of the nodes composing a software part sp .

Assumption A-M/A.

The larger the depth of a hierarchy, the larger the context information to be known in the lower nodes, the more likely the occurrence of error.

◇

Metric 7: Number of paths.

$No_Paths(sp)$ will denote the number of complete paths (from root to leaf) within a software part sp .

Assumption A-NOP.

The larger the number of paths, the larger the number of parent, sibling, and child relationships to be dealt with, the larger the complexity of the hierarchy, the higher the likelihood of error occurrence.

◇

2.2.4 Interaction-Based Metrics

In this section, we focus specifically on the dependencies that can propagate inconsistencies from data declarations to data declarations or subroutines when a new software part is integrated in a system. Those relationships will be called *interactions* and will be used to define metrics capturing cohesion and coupling within and between software parts, respectively. (Interactions linking subroutines to subroutines or data declarations will not be considered because they are, in the vast majority of cases, encapsulated in module or routine bodies and are therefore not detectable in our framework, which only takes into account high-level design.)

Definition 3: Data declaration-Data declaration (DD) Interaction.

A data declaration *A* DD-interacts with another data declaration *B* if a change in *A*'s declaration or use may cause the need for a change in *B*'s declaration or use.

◊

The DD-interaction relationship is transitive. If *A* DD-interacts with *B*, and *B* DD-interacts with *C*, then a change in *A* may cause a change in *C*, i.e., *A* DD-interacts with *C*. Data declarations can DD-interact with each other regardless of their location in the designed system. Therefore, the DD-interaction relationship can link data declarations belonging to the same software part or to different software parts.

The DD-interaction relationships can be defined in terms of the basic relationships between data declarations allowed by the language, which represent direct DD-interactions (i.e., not obtained by virtue of the transitivity of interaction relationships). Data declaration *A* directly DD-interacts with data declaration *B* if *A* is used in *B*'s declaration or in a statement where *B* is assigned a value. As a consequence, as bodies are not available at high-level design time, we will only consider interactions detectable from the interfaces.

DD-interactions provide a means to represent the dependency relationships between individual data declarations. Yet, DD-interactions *per se* are not able to capture the relationships between individual data declarations and subroutines, which are useful to understand whether data declarations and subroutines are related to each other and therefore should be encapsulated into the same module (see Section 2.2.4.1 on cohesion).

Definition 4: Data declaration-Subroutine (DS) Interaction.

A data declaration DS-interacts with a subroutine if it DD-interacts with at least one of its data declarations.

◊

Whenever a data declaration DD-interacts with *at least one* of the data declarations contained in a subroutine interface, the DS-interaction relationship between the data declaration and the subroutine can be detected by examining the high-level design. For instance, from the Ada-like code fragment in Figure 1, it is apparent that both type *T1* and object *OBJECT11* DS-interact with procedure *SR11*, since they both DD-interact with parameter *PAR11*, procedure *SR11*'s interface data declaration.

```
package M1 is
...
  type T1 is ...;
  OBJECT11, OBJECT12: T1;
  procedure SR11(PAR11: in T1:=OBJECT11);
...
  package M2 is
    ...
    OBJECT13: T1;
    type T2 is array (1..100) of T1;
    OBJECT21: T2;
    procedure SR21(PAR21: in out T2);
    ...
  end M2;
  ...
  OBJECT22: M2.T2;
  ...
end M1;
```

Figure 1. Program fragment

For graphical convenience, both sets of interaction relationships will be represented by directed graphs, the *DD-interaction graph*, and the *DS-interaction graph*, respectively. In both graphs (see Figure 2, which shows DD- and DS-interaction graphs for the code fragment of Figure 1), data declarations are represented by rounded nodes, subroutines by thick lined boxes, modules by thin lined boxes, and interactions by arcs.

Next, we will define high-level design metrics for cohesion and coupling, based on the above definitions. It is generally acknowledged that system architecture should have low coupling and high cohesion [CY79]. This is assumed to improve the capability of a system to be decomposed in highly independent and easy to understand pieces. However, the reader should bear in mind that high cohesion and low coupling may be conflicting goals, i.e., a trade-off between the two may exist. For instance, a software system can be made of small modules with a high degree of internal cohesion but very closely related to each other and, therefore, with a high level of coupling. Conversely, a software system can be composed of few large modules, representing its subsystems, loosely related to one another, i.e., with low coupling, but with a low degree of internal cohesion as well.

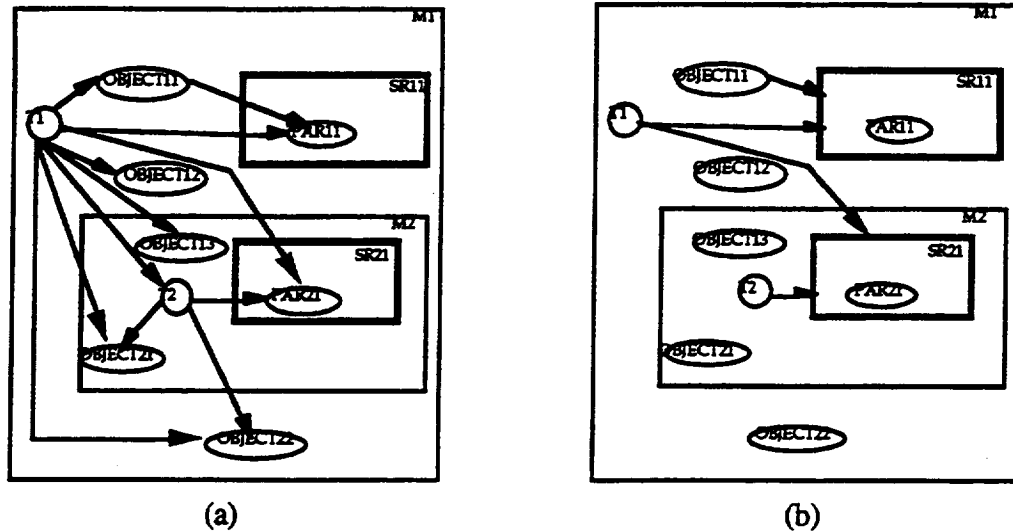


Figure 2. DD-interaction (a) and DS-interaction (b) graphs for the program fragment in Figure 1

Moreover, high cohesion and low coupling are not the only factors to be taken into account when designing a software system. Other issues (e.g., potential reuse) must be taken into account as well.

2.2.4.1 Cohesion

Cohesion captures the extent to which, in a software part, each group of data declarations and subroutines that are conceptually related belong to the same module. Based on

- an assumption (*A-CH*), which provides the rationale to define cohesion metrics;
- the concept of cohesive interactions, i.e., those interactions which contribute to cohesion;
- a set of properties (Properties 1-3) that cohesion metrics must have in order to measure cohesion

we now introduce a set of metrics (Metrics 8-11) to measure the degree of cohesion of a software part.

Assumption A-CH:

A high degree of cohesion is desirable because information related to declaration and subroutine dependencies should not be scattered across the system and among irrelevant

information. Data declarations and subroutines which are not related to each other should be encapsulated to the extent possible into different modules. As a result of such a strategy, we expect the software parts to be less error-prone.

◇

Consistently with the definition of Abstract Data Type/Object, data declarations and subroutines should show some kind of interaction between them, if they are conceptually related. Therefore, we are interested in evaluating the tightness of the interactions between the data declarations and data declarations or subroutines declared in a module interface. We will capture this by means of *cohesive* interactions.

Definition 5: Cohesive Interaction.

The set of cohesive interactions in a module m , denoted by $CI(m)$, is the union of the sets of DS-interactions and DD-interactions, with the exception of those DD-interactions between a data declaration and a subroutine formal parameter.

◇

We do not consider the DD-interactions linking a data declaration to a subroutine parameter as relevant to cohesion, since they are already accounted for by DS-interactions and we are interested in evaluating the degree of cohesion between data declarations and routines seen as a whole. Furthermore, cohesive interactions occur between data declarations and subroutines belonging to the same module. Interactions across different modules are not considered cohesive, since cohesion is the extent to which data declarations and subroutines that are conceptually related belong to the same module. Interactions across different modules contribute to coupling. Therefore, given a software part sp , the sets of cohesive interactions of its constituent modules (if any) are disjoint.

Remark.

It is worth reminding the reader that those relationships that cannot be detected by inspecting the interfaces, i.e., global variables interacting with subroutine bodies, can actually be quite relevant to cohesion evaluation, because they often represent the connections between an object and the subroutines that manipulate it. This issue will be further discussed later in this section.

◇

We base our cohesion metrics for software parts on cohesive interactions. Before defining them, we introduce the following three properties that they must satisfy in order to match our assumptions¹.

Property 1: Normalization.

Given a software part sp , the metric $cohesion(sp)$ belongs to a specified interval $[0, Max]$, and $cohesion(sp) = 0$ if and only if $CI(sp)$ is empty, and $cohesion(sp) = Max$ if and only if $CI(sp)$ includes all possible cohesive interactions.

◊

Normalization allows meaningful comparisons between the cohesions of different software parts, since they all belong to the same interval, and the extreme values of the cohesion range must represent the extreme cases. We will denote by $M(sp)$ the maximal set of cohesive interactions of the software part sp , i.e., the set that includes all of sp 's possible cohesive interactions, obtained by linking every data declaration to every other data declaration and subroutine with which it can interact. Some care must be used in defining $M(sp)$ for languages that allow circular type definitions, such as the ones used to define the nodes of a linked list. In this case, the declarations of two types $T1$ and $T2$ are built in such a way that $T1$ interacts with $T2$ and $T2$ interacts with $T1$. We choose to count only one interaction between them. This is explained by the fact that a single interaction between two data declarations justifies their encapsulation in a single module/Abstract Data Type.

Property 2: Monotonicity.

Let sp_1 be a software part and $CI(sp_1)$ its set of cohesive interactions. If sp_2 is a modified version of sp_1 with the same sets of data and subroutine declarations and one more cohesive interaction so that $CI(sp_2)$ includes $CI(sp_1)$, then $cohesion(sp_2) \geq cohesion(sp_1)$.

◊

Adding cohesive interactions to a software part cannot decrease its cohesion.

Property 3: Cohesive Modules.

Let sp be a software part, and let m_1 and m_2 be two of its modules. Let sp' be the software part obtained from sp by merging the declarations belonging to m_1 and m_2 into a new module m . If no cohesive interactions exist between the declarations belonging to m_1 and m_2 when they are grouped in m , then $cohesion(sp) \geq cohesion(sp')$.

◊

¹Properties and metrics can be defined for module sets more general than software parts. However, for simplicity, we will provide them only for software parts.

Splitting two sets of declarations which are not related to each other via cohesive interactions into two separate modules cannot decrease the cohesion of the software part.

Based on the properties defined above, we introduce a cohesion metric for software parts.

Metric 8: Ratio of Cohesive Interactions (RCI) for a Software Part.

The Ratio of Cohesive Interactions for sp is

$$RCI(sp) = \frac{|CI(sp)|}{|M(sp)|} \quad (*)$$

It is straightforward to prove that $RCI(sp)$ satisfies the above properties 1-3, and that, based on properties 1-3, it is defined on a ratio scale [F91]. Furthermore, $RCI(sp)$ can also be computed as a weighted sum of the $RCI(m)$'s of the single modules m belonging to sp . From Formula (*), since cohesive interactions only occur within modules, but not across modules

$$\begin{aligned} |CI(sp)| &= \sum_{m \in sp} |CI(m)| \\ |M(sp)| &= \sum_{n \in sp} |M(n)| \end{aligned}$$

so

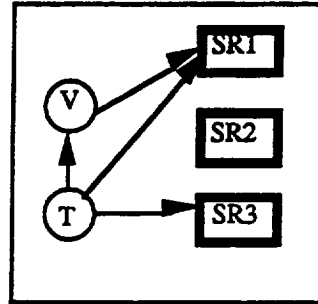
$$RCI(sp) = \sum_{m \in sp} \frac{|CI(m)|}{\sum_{n \in sp} |M(n)|}$$

By multiplying and dividing each term in the summation by $|M(m)|$, we obtain

$$RCI(sp) = \sum_{m \in sp} \frac{|M(m)|}{\sum_{n \in sp} |M(n)|} \frac{|CI(m)|}{|M(m)|} = \sum_{m \in sp} \frac{|M(m)|}{\sum_{n \in sp} |M(n)|} RCI(m)$$

The weights represent the potential contribution of each module m belonging to the software part sp to the cohesion of the whole sp .

Figure 3 shows an example of cohesion computation for a single module. T denotes a type declaration, C a variable declaration, and SR1, SR2, and SR3 subroutine declarations.



$$RCI = 4/7 = 0.571$$

Figure 3. Cohesion example

Based on the above cohesion metric, we can define a threshold for deciding whether a set of data and subroutines should be kept in one single module or divided into two or more modules. For simplicity, we will show here only the case in which we have to decide whether the declarations belonging to a module m should be split into two modules m_1 and m_2 . This should be the case if the cohesion of the software part consisting of the two modules m_1 and m_2 is greater than the cohesion of module m , i.e.,

$$\frac{|CI(m_1)| + |CI(m_2)|}{|M(m_1)| + |M(m_2)|} > \frac{|CI(m_1)| + |CI(m_2)| + |CI_{12}|}{|M(m)|}$$

where $|CI_{12}|$ is the number of cohesive interactions between the declarations belonging to modules m_1 and m_2 when they are in module m . Based on the above inequality, we can define a threshold on $|CI_{12}|$, as follows

$$\frac{(|M(m)| - |M(m_1)| - |M(m_2)|)(|CI(m_1)| + |CI(m_2)|)}{|M(m_1)| + |M(m_2)|} > |CI_{12}|$$

We want to emphasize, however, that, since cohesion is not the only characteristic relevant to software design, its increase should not be used as the *only* criterion on which to base such a decision.

The Role of Additional Information

Additional information to what is visible in the interfaces may be available at the end of high-level design. For instance, given the interface of a module *m*, the designers have at least a rough idea of which objects declared in *m* will be manipulated by a subroutine in *m*'s interface. It will be left to the person responsible for the metric program to decide whether or not it is worth collecting this kind of information, thus making the designer describe which objects will be accessed by which subroutines. Formatted comments may be a convenient way of conveying this information through module interfaces and therefore of automating the collection of this type of information.

For instance, from the code fragment in Figure 1, we cannot tell whether *OBJECT12* DS-interacts (as a global variable) with subroutine *SR11*. In this case, designers can answer in three different ways:

- (1) *OBJECT12* DS-interact with *P11*
- (2) *OBJECT12* does not DS-interact with *P11*
- (3) the information they have is not sufficient

It is worth saying that answers of kind (2) provide valuable, though negative, information on the DS-interactions present in a system. For instance, in the code fragment on Figure 1, the designer may indicate the existence of a DD-interaction between object *OBJECT12* and *PAR11* and the lack of interaction between *OBJECT13* and *PAR21*. As a consequence, the computation of cohesion is affected. If we take into account this additional information, other alternative cohesion metrics can be defined.

Given a software part *sp*, and a pair $\langle A, B \rangle$, where *A* is a data declaration and *B* is either a data declaration or a subroutine, we will say that the interaction between them is known if it is detectable from the high-level design or is signaled by the designers (they provide an answer similar to answer (1) above); we will say that the interaction between them is unknown if it is not detectable from the high-level design and is not signaled by the designers (they provide an answer similar to answer (1) above).

The set of known interactions of a software part *sp* will be denoted by $K(sp)$, and the set of unknown interactions by $U(sp)$. In general, $|M(sp)| \geq |K(sp)| + |U(sp)|$, since some interactions are not detectable from the high-level design and the designers explicitly exclude their existence.

Metric 9: Neutral Ratio of Cohesive Interactions (NRCI).

Unknown CIs are not taken into account

$$NRCI(sp) = \frac{|K(sp)|}{|M(sp)| - |U(sp)|}$$

Metric 10: Pessimistic Ratio of Cohesive Interactions (PRCI).

Unknown CIs are considered as if they were known not to be actual interactions.

$$PRCI(sp) = \frac{|K(sp)|}{|M(sp)|}$$

(This is equivalent to RCI(sp).)

Metric 11: Optimistic Ratio of Cohesive Interactions (ORCI).

Unknown CIs are considered as if they were known to be actual interactions

$$ORCI(sp) = \frac{|K(sp)| + |U(sp)|}{|M(sp)|}$$

The above three metrics satisfy Properties 1-3, where $CI(sp)$ is replaced by $K(sp) \cup U(sp)$.

If $PRCI(sp)$, $NRCI(sp)$, and $ORCI(sp)$ are all not undefined, it can be shown that, for all software parts sp ,

$$0 \leq PRCI(sp) \leq NRCI(sp) \leq ORCI(sp) \leq 1$$

$ORCI(sp)$ and $PRCI(sp)$ provide the bounds of the admissible range for cohesion, and $NRCI(sp)$ takes a value in between. It can also be shown that the smaller the number of unknown interactions, the smaller the interval $[PRCI, ORCI]$, i.e., the more complete the information, the narrower the uncertainty interval. It should be noted that, once the low-

level design is completed, accurate and complete information about cohesive interactions should be available.

Remark.

$\text{NRCI}(\text{sp})$ is undefined if and only if all interactions are unknown, i.e., no information is available on cohesive interactions. It is interesting to notice that in this case, and only in this case, $\text{PRCI}(\text{sp}) = 0$ and $\text{ORCI}(\text{sp}) = 1$, i.e., $\text{PRCI}(\text{sp})$ and $\text{ORCI}(\text{sp})$ do not provide stricter bounds than the ones provided by the interval for cohesion. The fact that $\text{NRCI}(\text{sp})$ is undefined can be interpreted as the possibility that $\text{NRCI}(\text{sp})$ can take any value in the interval $[0,1]$.

2.2.4.2 Coupling

In this section, we first give general definitions and assumptions on coupling, then, we present a set of metrics, and discuss the issue of genericity in the context of coupling. To address the particular issue of coupling, we will refer to the *import interactions* of a module m as all interactions going from a declaration outside m to a declaration inside m . Similarly, we define *export interactions* as going from a declaration located inside m to a declaration outside m .

Assumption A-IC:

The more dependent a software part on external data declarations, the more external information needs to be known in order to make the software part consistent with the rest of the system. In other words, the larger the amount of external data declarations, the more incomplete the local description of the software part interface, the more spread the information necessary to integrate a software part in a system. Thus, the software part becomes more error-prone.

◇

Definition 6: Import Coupling of a software part (IC).

Import Coupling is the extent to which a software part depends on imported external data declarations.

◇

Assumption A-EC:

Export coupling is related to how a software part is used in the system. The more often the software part is used, the larger the number of services it has to provide, the more flexible it needs to be, e.g., generic module. This may lead to errors.

◇

Definition 7: Export Coupling of a software part (EC).

Export coupling is the extent to which the data declarations of a software part affect the data declarations of the other software parts in the system.

◇

Import and export coupling of a software part will be expressed in terms of the actual DD-interactions involving imported external data declarations and the internal data declarations of the software part. We now provide properties that must be satisfied by both import and export coupling metrics.

Property 4: Non negativity

Given a software part sp , the metric $import_coupling(sp) \geq 0$ (resp. $export_coupling(sp) \geq 0$). $import_coupling(sp) = 0$ (resp. $export_coupling(sp) = 0$) if and only if sp does not have import (resp. export) interactions with other software parts.

◇

Property 5: Monotonicity

Let m_1 be a module and $II(m_1)$ (resp. $EI(m_1)$) its set of import (resp. export) interactions. If m_2 is a modified version of m_1 with the same sets of data and subroutine declarations and one more import (resp. export) interaction so that $II(m_2)$ (resp. $EI(m_2)$) includes $II(m_1)$ (resp. $EI(m_1)$), then $import_coupling(m_2) \geq import_coupling(m_1)$ (resp. $export_coupling(m_2) \geq export_coupling(m_1)$).

◇

Adding import (resp. export) interactions to a module cannot decrease its import (resp. export) coupling.

Property 6: Merging of Modules

The sum of the couplings of two modules is no less than the coupling of the module which is composed of the data declarations of the two modules.

◇

This stems from the fact that two modules may contain interactions between each other's declarations, which are no longer import or export interactions for the module resulting from merging the original modules.

It should be noted that, as opposed to cohesion, coupling is not a normalized metric. This comes from assumptions A-CH, A-IC, and A-EC (see Sections 2.2.4.1 and 2.2.4.2), where we state that cohesion is a *degree* of interdependence within a software part, whereas coupling is a *quantity* of dependencies between a software part and the rest of the system.

We will now introduce interaction-based coupling metrics. The issue will be first addressed by ignoring generic modules for the sake of simplification. Generic modules and their impact on the defined metrics will be treated later in this section.

Metric 12: Import Coupling

Given a software part sp , Import Coupling of sp (denoted by $IC(sp)$) is the number of DD-interactions between data declarations external to sp and the data declarations within sp .

Metric 13: Export Coupling

Given a software part sp , Export Coupling of sp (denoted by $EC(sp)$) is the number of DD-interactions between the data declarations within sp and the data declarations external to sp .

It is straightforward to prove that $IC(sp)$ and $EC(sp)$ satisfy the above properties 4-6, and that, based on properties 4-6, these metrics are defined on a ratio scale [F91].

Each box in Figure 4 represents a module interface. Module interfaces $m2$ and $m3$ are located in their parent's interface $m1$. $m2$ is assumed to be declared before $m3$ and therefore visible to $m3$. T_{ij} and $OBJECT_{ij}$ data declarations represent respectively types and objects in module m_i . $FP3$ represents a subroutine formal parameter. The IC and EC values for the modules in Figure 4 are computed as follows.

$IC(m1) = 0$	$EC(m1) = 10$
$IC(m2) = 4$	$EC(m2) = 1$
$IC(m3) = 5$	$EC(m3) = 0$
$IC(m4) = 2$	$EC(m4) = 0$

In the example of Figure 4, we see that $m1$ expectedly shows the largest export coupling.

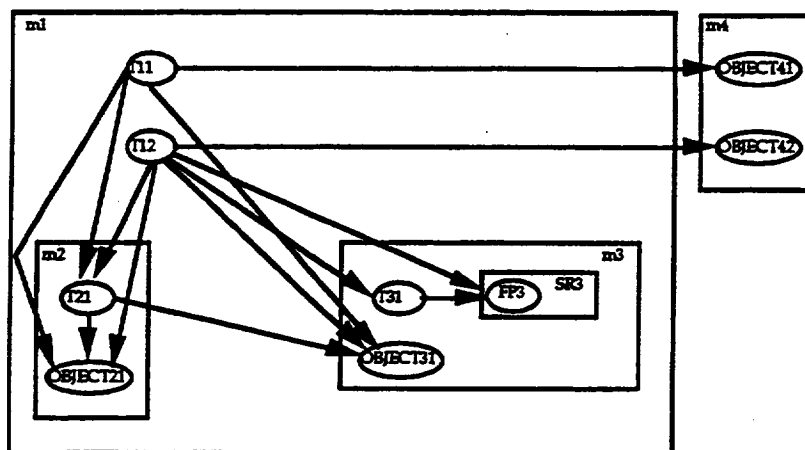


Figure 4. Calculation of IC and EC with non-generic modules only

Based on the definitions of $IC(sp)$ and $EC(sp)$, we derive four related metrics, $DIC(sp)$ (Direct Import Coupling), $TIC(sp)$ (Transitive Import Coupling), $DEC(sp)$ (Direct Export Coupling), $TEC(sp)$ (Transitive Export Coupling). $DIC(sp)$ and $DEC(sp)$ only take into account *direct* interactions, whereas $TIC(sp)$ and $TEC(sp)$ only take into account *transitive* interactions. By their definitions, $IC(sp) = DIC(sp) + TIC(sp)$, and $EC(sp) = DEC(sp) + TEC(sp)$. This allows us to separately evaluate the impact of direct and transitive interactions on error-proneness, as we show in the experimental validation. In practice, the number of transitive interactions turns out to be much bigger than that of direct interactions, so $IC(sp) \approx TIC(sp)$ and $EC(sp) \approx TEC(sp)$.

The Treatment of Generic Modules

There are two possible ways of taking into account generics when calculating coupling. Either each instance can be seen as a different module or a generic can be seen as any other module whose scope/global data declarations is/are the union of the scope/global data declarations of its instances. The second solution does not consider instances as independent modules and appears to be more suitable to our specific perspective, since errors are to be found in generics and, only as a consequence, in instances.

The import coupling of a generic module is the cardinality of the union of the sets of DD-interactions between the data declarations in the software system and those of each of its instances. When calculating export coupling, we take into account the DD-interactions between the data declarations of each of its instances and those of the software system. Consistent with the definition of DD-interaction, generic formal parameters DD-interact

with their particular generic actual parameters (i.e. type, object) when the generic module is instantiated, since a change in the former may imply a change in the latter.

This is what the example in Figure 5 illustrates. *Gen_m* is the interface of a generic module, with a generic formal parameter *GenFP* and a generic type *GenT*. The export coupling of module *Gen_m* is given by the sum of three parts

- two interactions from *Gen_m* to *m1*, due to the two instantiations, *Gen_m(1)* and *Gen_m(2)*, of *Gen_m* in *m1*,
- the interaction from the instantiation *Gen_m(1)*
- the two interactions from the instantiation *Gen_m(2)*.

$IC(m1) = 2$	$EC(m1) = 4$
$IC(m2) = 3$	$EC(m2) = 0$
$IC(m3) = 4$	$EC(m3) = 0$
$IC(Gen_m) = 0$	$EC(Gen_m) = 5$

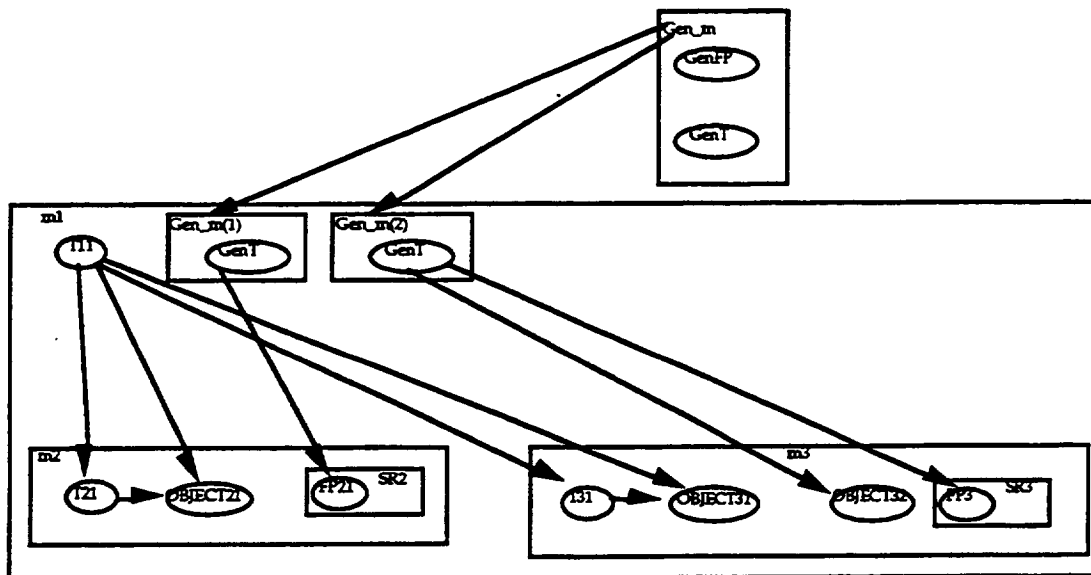


Figure 5. Generics when calculating coupling

3 Experimental Validation

The experimental validation has two main goals.

Goal 1

We want to find out which of the metrics defined above have a significant impact on the error-proneness of software parts. This allows us to

- a. prove that high-level design information can be used to build significant indicators of software error-proneness
- b. determine which of our assumptions about the development process (Section 2) are experimentally supported
- c. compare the four strategies for defining high-level design metrics
- d. identify the most promising research directions.

◊

Goal 2

We need to investigate dependencies between metrics, in order to determine which ones are complementary, and can be used in combination, and which ones capture similar phenomena.

◊

Section 3.1 presents the experimental design of the analysis, the project data sets used and the tool built to capture the discussed design metrics. Section 3.2 provides and discusses the results of a univariate analysis of the metrics. The significance of the metrics as predictors of error-prone software parts is assessed and the differences between systems are investigated. Section 3.3 investigates the results obtained when building multivariate classification models for detecting error-prone LMHs based on significant design metrics. The model results are assessed and the model functional structure is investigated.

3.1 Experiment Design

Experiment Layout

In order to validate software measurement assumptions experimentally, one can adopt two main strategies: (1) small-scale controlled experiments, (2) real-scale industrial case studies. In this research project, we chose the second alternative since we thought the

phenomena we are studying would be even more visible and significant on software systems of realistic size and complexity. Also, we thought that (2) should be a more relevant and convincing validation for the software industry practitioners.

However, the problem in such studies is that it becomes difficult to study the phenomena of interest in isolation, without having to deal with other sources of variation. In our case, we thought that, if these metrics were to be interesting, they should explain a significant percentage of the variation individually or in combination, despite other sources of variation. However, we expect some degree of variation across projects.

Environment

The first system studied is an attitude ground support software for satellites (GOADA) developed at the NASA Goddard Space Flight Center. The second one (GOESIM) is a dynamic simulator for a geostationary environmental satellite. These systems are composed of 525 and 676 Ada units, 90 Klocs and 170 Klocs, respectively, and have a fairly small reuse rate (around 5% of source code lines). The third system we studied (TONS) is an onboard navigation system for satellite that has been developed in the same environment and is about 180 Ada units and 50 Klocs large, with an extremely small rate of reuse (2% of source code lines). We selected projects with lower rates of reuse in order to make our analysis of design factors more straightforward by removing what we think is a major source of noise in this context.

Tool

A tool analyzing the interface parts of Ada source code has been developed in order to capture the design characteristics of these systems. This tool is based on LEX&YACC [LY92] and extracts generic high-level design information about the visibility and interactions of the system declarations. This information is consequently used to compute the metrics presented in Section 2.2, and others that might be defined.

Analytical Model

The response variable we use to validate the design metrics is binary, i.e., Did an error *not* occur in an LMH? In order to analyze the impact of software metrics on the error-proneness of software parts, we used logistic regression, a classification technique [HL89] used in many experimental sciences, based on maximum likelihood estimation, and presented below. In this case, a careful outlier analysis must be performed in order to make sure that

the observed trend is not the result of few observations [DG84]². In particular, we first used univariate logistic regression, to evaluate the impact of each of the metrics in isolation on error-proneness. Then, we performed multivariate logistic regression, to evaluate the relative impact of those metrics that had been assessed sufficiently significant in the univariate analysis (e.g., $\alpha < 0.20$ is a reasonable heuristic). This modeling process is further described in [HL89].

A multivariate logistic regression model is based on the following relationship equation (the univariate logistic regression model is a special case of this, where only one variable appears):

$$\log\left(\frac{p}{1-p}\right) = C_0 + C_1X_1 + C_2X_2 + \dots + C_nX_n$$

where p is the probability that no errors were found in a software part during the validation phase, and the X_i 's are the design metrics included as predictors in the model (called *covariates* of the logistic regression equation). In the two extreme cases, i.e., when a variable is either non-significant or entirely differentiates error-prone software parts, the curve (between p and any single X_i , i.e., assuming that all other X_j 's are constant) approximates a horizontal line and a vertical line respectively. In between, the curve takes a flexible S shape. However, since p is unknown, the coefficients C_i will be estimated through a likelihood function optimization. This procedure assumes that all observations are statistically independent. When building the regression equations, each observation was weighted according to the number of errors detected in each software part. The rationale is that each (non) detection of error is considered as an independent event. As a consequence, software parts where no errors were detected were weighted 1.

Goodness-of-fit for such a model is assessed via a statistic called R^2 (because similar in concept to the least-square regression coefficient of determination), belonging to the interval [0,1]. The higher R^2 , the more accurate the model. However, as opposed to the R^2 of least-square regression, high R^2 s are rare for logistic regression, for reasons whose explanation is well beyond the scope of this text. The interested reader may refer to [HL89] for a detailed introduction to logistic regression.

Tables 1 and 2 contain the results we obtained through, respectively, univariate and multivariate logistic regression on the three systems. We report those related to the metrics

²In addition, in order to confirm the obtained results, we used non-parametric tests for rank distributions such as the Mann-Whitney U test [CAP88]. Results appeared to be consistent across techniques and, in order to limit the amount of statistics provided to the reader and preserve the clarity of the text, we only show the results obtained with logistic regression.

that turned out to be the most significant ones across all three projects. For each metric, we provide the following statistics:

- C (appearing in both tables), the estimated regression coefficient. The larger the coefficient, the stronger the impact of the explanatory variable on the probability p .
- $\Delta\psi$ (appearing in Table 1 only), which is based on the notion of odd ratio [HL89], and provides an evaluation of the impact of the metric on the dependent variable. More specifically, the odd ratio $\psi(X)$ represents the ratio between the probability of not having an error and the probability of having an error when the value of the metric is X . As an example, if, for a given value X , $\psi(X)$ is 2, then it is twice more likely that the software part does not contain errors than that it does contain errors. The value of $\Delta\psi$ is computed by means of the following formula

$$\Delta\psi = \frac{\psi(X+1)}{\psi(X)}$$

Therefore, $\Delta\psi$ represents the reduction/increase in the odd ratio when the value X increases by 1 unit. This provides a more intuitive insight than regression coefficients into the impact of explanatory variables. (Since the whole range of RCI is $[0,1]$, we used one-tenth as the quantum for RCI increase with respect to which $\Delta\psi$ is computed.)

- α (appearing in both tables), the level of significance, which provides an insight into the accuracy of the coefficient estimates. The significance (α) of the logistic regression coefficients tells the reader about the probability for the coefficient to be different from zero by chance. Also, the larger the level of significance, the larger the standard deviation of the estimated coefficients, the less believable the calculated impact of the coefficient. The significance test is based on a likelihood ratio test [HL89] commonly used in the framework of logistic regression.

3.2 Univariate Analysis

Results

As Table 1 shows, all strategies presented in Section 2.2 provide significant metrics, but the strategy based on declaration counts. Therefore, these metrics, although based on simple and appealing concepts, do not appear to be significant predictors.

All the metrics based on exported declarations, i.e., *Local(sp)*, *ESP(sp)*, *EC(sp)*, *DEC(sp)*, and *TEC(sp)*, are not significant. Our explanation is that when an inconsistency exists between an exporting module *E* and an importing module *I*, *I* is more likely to be corrected, since *E* may export to other modules. Changing *E* is likely to require changing those other modules. Alternatively, a large amount of exports sometimes translates into a need for genericity but, for many declarations, just results into additional fields and dimensions. Therefore, the assumption underlying the export interactions metric appears somewhat questionable.

All the metrics based on the *IS_COMPONENT_OF* relation appear significant in the univariate analysis. However, they show a strong multicollinearity (i.e., the linear correlations are strong between metrics). Since *Avg_depth* is the best predictor in its category and in order to minimize the size of Table 1, only the *Avg_depth* results are shown.

A close analysis of the correlation matrix of the studied metrics shows that these results are not due to strong correlations between factors, e.g., when all factors are size predictors. Therefore, all the metrics in Table 1 seem to capture not only significant but different trends. This shows that most of the strategies are likely to be complementary and useful. This is confirmed by the multivariate results presented in Section 3.3.

Strategy	Project	GOADA			GOESIM			TONS		
	Metrics	C	$\Delta\psi$	α	C	$\Delta\psi$	α	C	$\Delta\psi$	α
USES	ISP	-0.8	45%	0.000	-0.717	49%	0.002	-0.96	38%	0.000
I_C_O	Avg_Depth	-2.27	11%	0.000	-2.4	9%	0.000	-3.9	2%	0.000
Inter.	RCI	0.63	188%	0.000	0.215	124%	0.047	0.34	141%	0.001
Inter.	TIC	-0.016	98.5%	0.001	-0.017	98.3%	0.002	-0.02	98%	0.15
Inter.	DIC	-0.23	79%	0.000	-0.19	83%	0.001	-0.04	96%	0.19

Table 1. Univariate Analysis

Detailed Discussion

TIC and *DIC* do not appear to be significant in *TONS* ($\alpha = 0.19$ and 0.15 , respectively), whereas they are very significant in the two other systems. The analysis of the distribution of these factors in all three systems, respectively, shows that their standard deviation (σ) and median (m) are much smaller in *TONS*, i.e., with respect to *TIC*, $\sigma = 10$, $m = 2.5$ for *TONS* versus $\sigma = 32.74$, $m = 15.5$ for *GOADA*, $\sigma = 32.18$, $m = 59$ for *GOESIM*. As a consequence, any trend related to either *DIC* or *TIC* is very likely not to be visible in the *TONS* dataset. When considering that *TONS* is a significantly smaller system than the two other ones, results may be interpreted as follows: the distribution of import interactions is strongly dependent on the size of the system and input interaction metrics are likely to be mediocre predictors for small systems.

Comparing Models

From a more general perspective, variations across models (i.e., univariate regression equations) should be expected due to differences in project characteristics, i.e., size, application domain. However, it is worth noticing that, despite the fact that these projects belong to different application domains (within the context of satellite support systems) and have been developed at different times, most of the models are surprisingly stable across projects. Because of the functional shape of logistic models, coefficients that may appear significantly different actually generate very similar models, e.g., In Table 1, coefficients -2.27 and -3.9 yield $\Delta\psi$'s of 11% and 2% , respectively. As a consequence, to evaluate the stability of the models, the reader should rather look at the $\Delta\psi$ column in Table 1. When doing so, only *RCI* appears to have a noticeable model instability even though the trends are consistent.

3.3 Multivariate Models

In this section, we present the results obtained by performing a stepwise multivariate logistic regression. Table 2 provides the estimated regression coefficients (C) and their significance (α) based on a *Wald test* [HL89], which is obtained by comparing the maximum likelihood estimate of a parameter to its estimated standard deviation. Regression coefficients are not shown when their level of significance is above 0.2 (substituted by a $*$).

Strategy	Projects	GOADA		GOESIM		TONS	
	Metrics	C	α	C	α	C	α
USES	ISP	-0.9	0.04	*	*	-1.18	0.000
I_C_O	Avg_Depth	-1.8	0.003	-3.12	0.000	-5.62	0.000
Inter.	RCI	0.4	0.006	0.3	0.07	0.2	0.16
Inter.	TIC	-0.023	0.000	-0.02	0.005	*	*
Inter.	DIC	0.23	0.04	-0.13	0.04	-0.11	0.002

Table 2. Coefficients of Multivariate Models

Results

The very low levels of significance in Table 2 suggest that these metrics may be used in combination as indicators of error-prone LMHs. Indeed, when used in a multivariate model, many of these metrics are still significant and produce models that are more accurate than univariate models (Table 2). The best univariate R^2 s are 0.115, 0.20 and 0.16 for *GOADA*, *GOESIM*, and *TONS*, respectively. In the same order, the multivariate R^2 s are 0.21, 0.24, and 0.43. We can see that the results improved very significantly for *GOADA* and *TONS*.

Interaction-based metrics are more complex but worth collecting, since they are the only metrics defined at the declaration level that appeared significant. In addition, the average LMH depth was consistently selected as a very good indicator. This is likely to be an early measure of "size" of the LMH and is expectedly significant. Also, *ISP*, a metric similar to the notion of fan-in shows to be significant across projects (except in the multivariate *GOESIM* model for reasons explained below), while *ESP*, the equivalent measure for exports (based on the fan-out of LMHs) is not significant. As a consequence, a metric of the form $(\text{fan_in} \cdot \text{fan_out})^2$, suggested in numerous occasions in the literature [HK84, IS88, S90, Z91], does not appear to be significant. From a more general perspective, metrics based on imports, regardless of the associated concepts, appear to predict more accurately the error-proneness of software parts.

Comparing Models

Some variability in the estimated regression coefficients can be observed across projects in Table 2 and requires some discussion. In multivariate models, coefficients have

a tendency to adjust, statistically, for other variables [HL89]. Sometimes, variables are weak predictors of the response variable when taken individually, and become more significant when integrated in a multivariate model. In Table 1, *DIC* showed, for *TONS*, a mediocre level of significance, whereas it appears to be a significant predictor in Table 2. Moreover, its coefficient is very unstable across projects and the trend is reversed (positive) for *GOADA* and *TONS*. When looking more carefully at the associations (not only the narrower concept of linear correlation) between metrics, it can be determined that this is the results of strong association between *DIC* and *ISP* in *GOADA* and *TONS*. These associations are a typical source of coefficient instability [DG84], e.g., the coefficient of *ISP* in *GOADA* varies from -0.9 to -0.39 when *DIC* is removed from the equation.

TIC remains non-significant because of its strong linear correlation ($R^2 = 0.76$) with *DIC* in the *TONS* dataset. Similarly, *ISP* does not appear significant in the *GOESIM* dataset because of a strong correlation with *DIC* ($R^2 = 0.50$). *RCI* in *TONS* shows a weaker significance ($\alpha = 0.16$) than in the univariate results and no strong linear correlation can be observed with the other metrics included in the multivariate equation. However, LMHs with large numbers of imported interactions are all located in the low part of the cohesion range. Such an *association* (likely to be spurious since it is not the case in the other datasets) with *DIC* is likely to affect the significance of *RCI* in a multivariate equation.

It is important to note that a different set of systems showing different distributions might show very different trends. This points out a need for large scale investigation across various development environments and application domains.

4 Conclusion

This study has shown that statistical models of extremely good significance can be built based on *high-level* design information. In particular, we have determined accurate early predictors for error-prone software. Moreover, the results suggested that, at this stage of understanding, several strategies were worth investigating because none of them showed dominant trends, while most of them appeared to be complementary. In order to provide the practitioner with usable, well understood and validated models, software engineering researchers will have to keep refining and validating the existing metrics. There is still substantial room for improvement.

The stability of the impact of these metrics across projects allows us to draw optimistic conclusions about the use of such quality indicators. Using early quality

indicators based on objective empirical evidence appears possible. However, it is very likely that this kind of indicators will behave differently across various domains of application and development environments.

Therefore, the use of such indicators should always be preceded by a careful empirical analysis of local error patterns and a thorough comparison across projects.

Our future work will be three-fold:

- Analyze more systems
- Validate further and refine the metrics we defined in this paper. The variations across environments and the study/comparison of different architectures is likely to give us interesting insights.
- Consistent with our current objectives, we will address the issues related to building metric based empirical models earlier in the life cycle. In particular, the next stage of this research will focus on defining and validating metrics for formal specifications.

Acknowledgments

We thank Giuseppe Calavaro and Chris Lott for their helpful comments on the earlier drafts of this paper.

REFERENCES

- [AE92] W. Agresti and W. Evanco, "Projecting Software Defects from Analyzing Ada Designs," *IEEE Trans. Software Eng.*, 18 (11), November, 1992.
- [BBH93] L. Briand, V. Basili and C. Hetmanski, "Developing Interpretable Models with Optimized Set Reduction for Identifying High Risk Software Components," *IEEE Trans. Software Eng.*, 19 (11), November, 1993.
- [BO87] G. Booch, "Software Engineering with Ada," Benjamin/Cumming Publishing Company, Inc., Menlo Park, California, 1987.
- [BR88] V. Basili and H. Rombach, "The TAME Project: Towards Improvement-Oriented Software Environments," *IEEE Trans. Software Eng.*, 14 (6), June, 1988.

- [BTH93] L. Briand, W. Thomas and C. Hetmanski, "Modeling and Managing Risk early in Software Development," *International Conference on Software Engineering*, Maryland, May 1993
- [CAP88] J. Capon, "Statistics for the Social Sciences", Wadworth publishing company, 1988
- [CY79] L. Constantine, E. Yourdon, "Structured Design," Prentice Hall, 1979
- [DG84] W. Dillon and M. Goldstein, *Multivariate Analysis: Methods and Applications*, Wiley and Sons, 1984.
- [DoD83] ANSI/MIL-STD-1815A-1983, Reference Manual of the Ada Programming Languages, U.S. Department of Defense, 1983
- [F91] Norman Fenton, "Software Metrics, A Rigorous Approach," Chapman&Hall, 1991.
- [G86] J. Gannon, E. Katz, V. Basili, "Metrics for Ada Packages: an Initial Study," *Communications of the ACM*, Vol. 29, N. 7, July 1986.
- [GJM92] C. Ghezzi, M. Jazayeri, D. Mandrioli, "Fundamentals of Software Engineering," Prentice Hall, Englewood Cliffs, NJ, 1992
- [HK84] S. Henry, D. Kafura, "The Evaluation of Systems' Structure Using Quantitative Metrics," *Software Practice and Experience*, 14 (6), June, 1984.
- [IS88] D. Ince, M. Shepperd, "System Design Metrics: a Review and Perspective," *Proc. Software Engineering 88*, pages 23-27, 1988
- [K88] B. Kitchenham, "An Evaluation of Software Structure Metrics," *Proc. COMPSAC 88*, 1988
- [LY92] J. Levine, T. Mason, D. Brown, "lex & yacc," O'Reilly & Associates, Inc., 1992
- [M77] J. Myers, "An Extension to the Cyclomatic Measure of Program Complexity," *SIGPLAN Notices*, 12(10):61-64, 1977
- [MGBB90] A. Melton, D. Gustafson, J. Bieman, A. Baker, "A Mathematical Perspective for Software Measures Research," *Software Engineering Journal*, September 1990.
- [R87] H. D. Rombach, "A Controlled Experiment on the Impact of Software Structure and Maintainability:," *IEEE Trans. Software Eng.*, 13 (5), May, 1987.
- [R90] H. D. Rombach, "Design Measurement: Some Lessons Learned," *IEEE Software*, March 1990.
- [S90] M. Shepperd, "Design Metrics: An Empirical Analysis," *Software Engineering Journal*, January 1990.

- [SB91] R. Selby and V. Basili, "Analyzing Error-Prone System Structure," *IEEE Trans. Software Eng.*, 17 (2), February, 1991.
- [Z91] W. Zage, D. Zage, P. McDaniel, I. Khan, "Evaluating Design Metrics on Large-Scale Software," SERC-TR-106-P, September 1991.

SECTION 3—TECHNOLOGY EVALUATIONS

The technical papers included in this section were originally prepared as indicated below.

- *Comparing Detection Methods for Software Requirements Inspections: A Replicated Experiment*, A. A. Porter, L. G. Votta Jr., and V. R. Basili, University of Maryland, Technical Report TR-3327, July 1994
- “Software Process Evolution at the SEL,” V. Basili, S. Green, *IEEE Software*, July 1994

Comparing Detection Methods For Software Requirements Inspections: A Replicated Experiment

Adam A. Porter Lawrence G. Votta, Jr. Victor R. Basili*

Abstract

Software requirements specifications (SRS) are usually validated by inspections, in which several reviewers independently analyze all or part of the specification and search for defects. These defects are then collected at a meeting of the reviewers and author(s).

Usually, reviewers use Ad Hoc or Checklist methods to uncover defects. These methods force all reviewers to rely on nonsystematic techniques to search for a wide variety of defects. We hypothesize that a Scenario-based method, in which each reviewer uses different, systematic techniques to search for different, specific classes of defects, will have a significantly higher success rate.

We evaluated this hypothesis using a 3×2^4 partial factorial, randomized experimental design. Forty eight graduate students in computer science participated in the experiment. They were assembled into sixteen, three-person teams. Each team inspected two SRS using some combination of Ad Hoc, Checklist or Scenario methods.

For each inspection we performed four measurements: (1) individual defect detection rate, (2) team defect detection rate, (3) percentage of defects first identified at the collection meeting (meeting gain rate), and (4) percentage of defects first identified by an individual, but never reported at the collection meeting (meeting loss rate).

The experimental results show that (1) the Scenario method has a higher defect detection rate than either Ad Hoc or Checklist methods, (2) Scenario reviewers are more effective at detecting the defects their scenarios are designed to uncover, and are no less effective at detecting other defects, (3) Checklist reviewers were no more effective than Ad Hoc reviewers, and (4) Collection meetings produce no net improvement in the defect detection rate - meeting gains are offset by meeting losses.

A preliminary version of this article entitled, "An Experiment to Assess Different Defect Detection Methods For Software Requirements Inspections", has been selected to appear in the proceedings of the 16th International Conference on Software Engineering. This article expands on our previous work in several ways:

1. We have replicated the initial experiment - doubling the number of data points.
2. We have expanded the description of the Scenario detection methods and included appendices containing the full text of the Ad Hoc, Checklist, and Scenario defect detection aids that were used during the experiment.
3. Our original analysis analyzed the effect of different detection methods on team performance. With the increased number of data points, we are now able to extend the analysis to determine how these methods influence individual performance. This allows us to reject several threats to the experiment's internal validity.
4. We have added a new section analyzing the how inspection meetings affect inspection performance. Our results show that meetings contribute nothing to defect detection effectiveness.

*This work is supported in part by the National Aeronautics and Space Administration under grant NSG-5123. Porter and Basili are with the Department of Computer Science, University of Maryland, College Park, Maryland 20472. Votta is with the Software Production Research Department, AT&T Bell Laboratories Naperville, IL 60566

1 Introduction

One of the most common ways of validating a software requirements specification (SRS) is to submit it to an inspection by a team of reviewers. Many organizations use a three-step inspection procedure for eliminating defects¹: detection, collection, and repair². [8, 17] A team of reviewers reads the SRS, identifying as many defects as possible. Newly identified defects are collected, usually at a team meeting, and then sent to the document's authors for repair.

We are focusing on the methods used to perform the first step in this process, defect detection. For this article, we define a defect detection method to be a set of defect detection techniques coupled with an assignment of responsibilities to individual reviewers.

Defect detection techniques may range in prescriptiveness from intuitive, nonsystematic procedures, such as Ad Hoc or Checklist techniques, to explicit and highly systematic procedures, such as formal proofs of correctness.

A reviewer's individual responsibility may be general – to identify as many defects as possible – or specific – to focus on a limited set of issues such as ensuring appropriate use of hardware interfaces, identifying untestable requirements, or checking conformity to coding standards.

These individual responsibilities may be coordinated among the members of a review team. When they are not coordinated, all reviewers have identical responsibilities. In contrast, the reviewers in coordinated teams may have separate and distinct responsibilities.

In practice, reviewers often use Ad Hoc or Checklist detection techniques to discharge identical, general responsibilities. Some authors, notably Parnas and Weiss[13], have argued that inspections would be more effective if each reviewer used a different set of systematic detection techniques to discharge different, specific responsibilities.

Until now, however, there have been no reproducible, quantitative studies comparing alternative detection methods for software inspections. We have conducted such an experiment and our results demonstrate that the choice of defect detection method significantly affects inspection performance. Furthermore, our experimental design may be easily replicated by interested researchers.

¹We use the word *defect* instead of the word *fault* even though this does not adhere to the IEEE Standards on Software Engineering Terminology [9]. We feel the word *fault* has a code-specific connotation – only one of the many places where inspections are used.

²Depending on the exact form of the inspection, they are sometimes called reviews or walkthroughs. For a more thorough description of the taxonomy see [8] pp. 171ff and [10].

Below we describe the relevant literature, several alternative defect detection methods which motivated our study, our research hypothesis, and our experimental observations, analysis and conclusions.

1.1 Inspection Literature

A summary of the origins and the current practice of inspections may be found in Humphrey [8]. Consequently, we will discuss only work directly related to our current efforts.

Fagan[6] defined the basic software inspection process. While most writers have endorsed his approach[3, 8], Parnas and Weiss are more critical [13]. In part, they argue that effectiveness suffers because individual reviewers are not assigned specific responsibilities and because they lack systematic techniques for meeting those responsibilities.

Some might argue that Checklists are systematic because they help define each reviewer's responsibilities and suggest ways to identify defects. Certainly, Checklists often pose questions that help reviewers discover defects. However, we argue that the generality of these questions and the lack of concrete strategies for answering them makes the approach nonsystematic.

To address these concerns – at least for software designs – Parnas and Weiss introduced the idea of active design reviews. The principal characteristic of an active design review is that each individual reviewer reads for a specific purpose, using specialized questionnaires. This proposal forms the motivation for the detection method proposed in Section 2.2.2.

1.2 Detection Methods

Ad Hoc and Checklist methods are the two most frequently used defect detection methods. With Ad Hoc detection methods, all reviewers use nonsystematic techniques and are assigned the same general responsibilities.

Checklist methods are similar to Ad Hoc, but each reviewer receives a checklist. Checklist items capture important lessons learned from previous inspections within an environment or application. Individual checklist items may enumerate characteristic defects, prioritize different defects, or pose questions that help reviewers discover defects, such as “Are all interfaces clearly defined?” or “If input is received at a faster rate than can be processed, how is this handled?” The purpose of these items is to focus reviewer responsibilities and suggest ways for reviewers to identify defects.

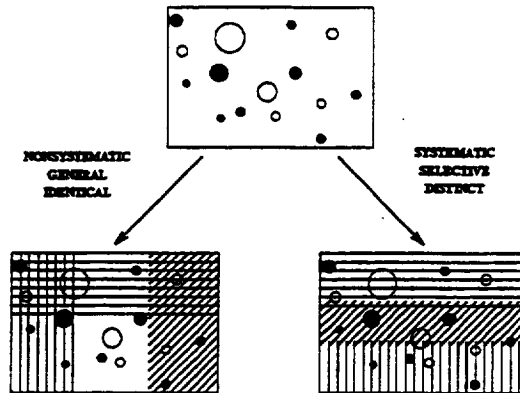


Figure 1: **Systematic Inspection Research Hypothesis.** This figure represents a software requirements specification before and after a *nonsystematic* technique, *general* and *identical* responsibility inspection and a *systematic* technique, *specific* and *distinct* responsibility inspection. The points and holes represent various defects. The line-filled regions indicate the coverage achieved by different members of the inspection team. Our hypothesis is that systematic technique, specific and coordinated responsibility inspections achieve broader coverage and minimize reviewer overlap, resulting in higher defect detection rates and greater cost benefits than nonsystematic methods.

1.3 Hypothesis

We believe that an alternative approach which gives individual reviewers specific, orthogonal detection responsibilities and specialized techniques for meeting them will result in more effective inspections.

To explore this alternative we developed a set of defect-specific techniques called Scenarios – collections of procedures for detecting particular classes of defects. Each reviewer executes a single scenario and multiple reviewers are coordinated to achieve broad coverage of the document.

Our underlying hypothesis is depicted in Figure 1: that nonsystematic techniques with general reviewer responsibility and no reviewer coordination, lead to overlap and gaps, thereby lowering the overall inspection effectiveness; while systematic approaches with specific, coordinated responsibilities reduce gaps, thereby increasing the overall effectiveness of the inspection.

2 The Experiment

To evaluate our systematic inspection hypothesis we designed and conducted a multi-trial experiment. The goals of this experiment were twofold: to characterize the behavior of existing approaches and to assess the potential benefits of Scenario-based methods. We ran the experiment twice; once in the Spring of 1993, and once the following Fall. Both runs used 24 subjects – students taking a graduate course in formal methods who acted

as reviewers. Each complete run consisted of (1) a training phase in which the subjects were taught inspection methods and the experimental procedures, and in which they inspected a sample SRS, and (2) an experimental phase in which the subjects conducted two monitored inspections.

2.1 Experimental Design

The design of the experiment is somewhat unusual. To avoid misinterpreting the data it is important to understand the experiment and the reasons for certain elements of its design ³.

2.1.1 Variables

The experiment manipulates five independent variables:

1. the detection method used by a reviewer (Ad Hoc, Checklist, or Scenario);
2. the experimental replication (we conducted two separate replications);
3. the inspection round (each reviewer participates in two inspections during the experiment);
4. the specification to be inspected (two are used during the experiment).
5. the order in which the specifications are inspected (either specification can be inspected first).

The detection method is our treatment variable. The other variables allow us to assess several potential threats to the experiment's internal validity.

For each inspection we measure four dependent variables:

1. the individual defect detection rate,
2. the team defect detection rate ⁴,
3. the percentage of defects first identified at the collection meeting (meeting gain rate), and
4. the percentage of defects first identified by an individual, but never reported at the collection meeting (meeting loss rate).

³See Judd, et al. [11], chapter 4 for an excellent discussion of randomized social experimental designs.

⁴The team and individual defect detection rates are the number of defects detected by a team or individual divided by the total number of defects known to be in the specification. The closer that value is to 1, the more effective the detection method. No defects were intentionally seeded into the specifications. All defects are naturally occurring.

Detection Method	Round/Specification			
	Round 1		Round 2	
	WLMS	CRUISE	WLMS	CRUISE
ad hoc	1B, 1D, 1G 1H, 2A	1A, 1C, 1E 1F, 2D	1A	1D, 2B
checklist	2B	2E, 2G	1E, 2D, 2G	1B, 1H
scenarios	2C, 2F	2H	1F, 1C, 2E 2H	1G, 2A, 2C 2F

Table 1: This table shows the settings of the independent variables. Each team inspects two documents, the WLMS and CRUISE, one per round, using one of the three detection methods. Teams from the first replication are denoted 1A–1H, teams from the second replication are denoted 2A–2H.

2.1.2 Design

The purpose of this experiment is to compare the Ad Hoc, Checklist, and Scenario detection methods for inspecting software requirements specifications.

When comparing multiple treatments, experimenters frequently use fractional factorial designs. These designs systematically explore all combinations of the independent variables, allowing extraneous factors such as team ability, specification quality, and learning to be measured and eliminated from the experimental analysis.

Had we used such a design each team would have participated in three inspection rounds, reviewing each of three specifications and using each of three methods exactly once. The order in which the methods are applied and the specifications are inspected would have been dictated by the experimental design.

Such designs are unacceptable for this study because they require some teams to use the Ad Hoc or Checklist method after they have used the Scenario method. Since the Ad Hoc and Checklist methods are nonsystematic, it is impossible to define, monitor and enforce their use. Therefore, we were concerned that the use of the Scenario method in an early round might imperceptibly distort the use of the other methods in later rounds.

Consequently, we chose a partial factorial design in which not all combinations of the independent variables are present. With this design, each team participates in two inspections, using some combination of the three detection methods, but teams using the Scenario method in the first round must continue to use it in the second round. Table 1 shows the settings of the independent variables.

2.1.3 Threats to Internal Validity

A potential problem in any experiment is that some factor may affect the dependent variable without the researcher's knowledge. This possibility must be minimized. We considered five such threats: (1) selection effects,

(2) maturation effects, (3) replication effects, (4) instrumentation effects, and (5) presentation effects.

Selection effects are due to natural variation in human performance. For example, random assignment of subjects may accidentally create an elite team. Therefore, the difference in this team's natural ability will mask differences in the detection method performance. Two approaches are often taken to limit this effect:

1. Create teams with equal skills. For example, rate each participant's background knowledge and experience as either low, medium, or high and then form teams of three by selecting one individual at random from each experience category. Detection methods are then assigned to fit the needs of the experiment.
2. Compose teams randomly, but require each team to use all three methods. In this way, differences in team skill are spread across all treatments.

Neither approach is entirely appropriate. Although, we used the first approach in our initial replication, the approach is unacceptable for multiple replications, because even if teams within a given replication have equal skills, teams from different replications will not.

As discussed in the previous section, the second approach is also unsuitable because using the Scenarios in the first inspection Round will certainly bias the application of the Ad Hoc or Checklist methods in the second inspection Round.

Our strategy for the second replication and future replications is to randomly assign teams and detection methods. However, teams that used Scenarios in the first round were constrained to use them again in the second round. This compromise efficiently employs the subjects without biasing the performance of any teams.

Maturation effects are due to subjects learning as the experiment proceeds. We have manipulated the detection method used and the order in which the documents are inspected so that the presence of this effect can be discovered and taken into account.

Replication effects are caused by differences in the materials, participants, or execution of multiple replications. We limit this effect by using only first and second year graduate students as subjects - rather than both undergraduate and graduate students. Also, we maintain consistency in our experimental procedures by packaging the experimental procedures as a classroom laboratory exercise. This helps us to ensure that similar steps are followed for all replications.

As will be shown in Section 3, variation in the defect detection rate is not explained by selection, maturation,

or replication effects.

Finally, instrumentation effects may result from differences in the specification documents. Such variation is impossible to avoid, but we controlled for it by having each team inspect both documents.

2.1.4 Threats to External Validity

Threats to external validity limit our ability to generalize the results of our experiment to industrial practice. We identified three such threats:

1. the reviewers in the first run of our experiment may not be representative of software programming professionals;
2. the specification documents may not be representative of real programming problems;
3. the inspection process in our experimental design may not be representative of software development practice.

The first two threats are real. To surmount them we are currently replicating our experiment using software programming professionals to inspect industrial work products. Nevertheless, laboratory experimentation is a necessary first step because it greatly reduces the risk of transferring immature technology.

We avoided the third threat by modeling the experiment's inspection process after the design inspection process described in Eick, et al. [5], which is used by several development organizations at AT&T; therefore, we know that at least one professional software development organization practices inspections in this manner.

2.1.5 Analysis Strategy

Our analysis strategy had two steps. The first step was to find those independent variables that individually explain a significant amount of the variation in the team detection rate. This was done by using an analysis of variance technique as discussed in Box, et al. ([4], pp. 165ff).

The second step was to evaluate the combined effect of the variables shown to be significant in the initial analysis. Again, we followed Box, et al. closely ([4], pp. 210ff).

Once these relationships were discovered and their magnitude estimated, we examined other data, such as correlations between the categories of defects detected and the detection methods used that would confirm or

reject (if possible) a causal relationship between detection methods and inspection performance.

2.2 Experiment Instrumentation

We developed several instruments for this experiment: three small software requirements specifications (SRS), instructions and aids for each detection method, and a data collection form.

2.2.1 Software Requirements Specifications

The SRS we used describe three event-driven process control systems: an elevator control system, a water level monitoring system, and an automobile cruise control system. Each specification has four sections: Overview, Specific Functional Requirements, External Interfaces, and a Glossary. The overview is written in natural language, while the other three sections are specified using the SCR tabular requirements notation [7].

For this experiment, all three documents were adapted to adhere to the IEEE suggested format [10]. All defects present in these SRS appear in the original documents or were generated during the adaptation process; no defects were intentionally seeded into the document. The authors discovered 42 defects in the WLMS SRS; and 26 in the CRUISE SRS. The authors did not inspect the ELEVATOR SRS since it was only used for training exercises.

Elevator Control System (ELEVATOR) [18] describes the functional and performance requirements of a system for monitoring the operation of a bank of elevators (16 pages).

Water Level Monitoring System (WLMS) [16] describes the functional and performance requirements of a system for monitoring the operation of a steam generating system (24 pages).

Automobile Cruise Control System (CRUISE) [12] describes the functional and performance requirements for an automobile cruise control system (31 pages).

2.2.2 Defect Detection Methods

To make a fair assessment of the three detection methods (Ad Hoc, Checklist, and Scenario) each method should search for a well-defined population of defects. To accomplish this, we used a general defect taxonomy to define the responsibilities of Ad Hoc reviewers.

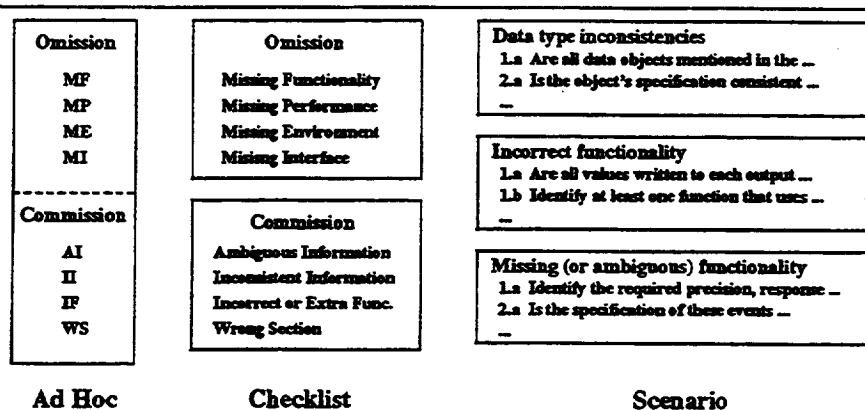


Figure 2: Relationship Between Defect Detection Methods. The figure depicts the relationship between the defect detection methods used in this study. The vertical extent represents the coverage. The horizontal axis labels the method and represents the degree of detail (the greater the horizontal extent the greater the detail). Moving from Ad Hoc to Checklist to Scenario there is more detail and less coverage. The gaps in the Scenario and Checklist columns indicate that the Checklist is a subset of the Ad Hoc and the Scenarios are a subset of the Checklist.

The checklist used in this study is a refinement of the taxonomy. Consequently, Checklist responsibilities are a subset of the Ad Hoc responsibilities.

The Scenarios are derived from the checklist by replacing individual Checklist items with procedures designed to implement them. As a result, Scenario responsibilities are distinct subsets of Checklist and Ad Hoc responsibilities. The relationship between the three methods is depicted in Figure 2.

The taxonomy is a composite of two schemes developed by Schneider, et al. [14] and Basili and Weiss [2]. Defects are divided into two broad types: omission – in which important information is left unstated and commission – in which incorrect, redundant, or ambiguous information is put into the SRS by the author. Omission defects were further subdivided into four categories: Missing Functionality, Missing Performance, Missing Environment, and Missing Interface. Commission defects were also divided into four categories: Ambiguous Information, Inconsistent Information, Incorrect or Extra Functionality, and Wrong Section. (See Appendix A for complete taxonomy.) We provided a copy of the taxonomy to each reviewer.

Ad Hoc reviewers received no further assistance.

Checklist reviewers received a single checklist derived from the defect taxonomy. To generate the checklist we populated the defect taxonomy with detailed questions culled from several industrial checklists. Thus, they are very similar to checklists used in practice. All Checklist reviewers used the same checklist. (See Appendix B for the complete checklist.)

Defect Report Form			
Specification	WLM5	Date	4/12
Team ID	C	Rev. ID	12
		Time In	1:40 PM
		Time Out	3:40
Defect No.		Activity (Read/Col.)	
Location(s)		Disposition (T/F)	
<div style="font-family: monospace; font-size: 0.9em;"> 6:12 The initialization of the variable <i>Translating %</i> causes an incorrect transition into a <i>Y</i> <i>value mode.</i> </div>			
Defect No.		Activity (Read/Col.)	
Location(s)		Disposition (T/F)	
<div style="font-family: monospace; font-size: 0.9em;"> 14:15 The "exp. water indicator" is allowed to have both the "on" and "off" values when the system is in test mode for between 2 and 4 seconds. </div>			

Figure 3: Reviewer Defect Report Form. This is a small sample of the defect report form completed during each reviewer's defect detection. Defects number 10 and 11, found by reviewer 12 of team C for the WLM5 specification are shown.

Finally, we developed three groups of Scenarios. Each group of Scenarios was designed for a specific subset of the Checklist items:

1. Data Type Inconsistencies (DF),
2. Incorrect Functionalities (IF),
3. Missing or Ambiguous Functionalities (MF).

After the experiment was finished we applied the Scenarios to estimate how broadly they covered the WLM5 and CRUISE defects. We estimated that the Scenarios address about half of the defects that are covered by the Checklist. Appendix C contains the complete list of Scenarios.

2.2.3 Defect Report Forms

We also developed a Defect Report Form. Whenever a potential defect was discovered – during either the defect detection or the collection activities – an entry was made on the form. The entry included four kinds of information: Inspection Activity (Detection, Collection); Defect Location (Page and Line Numbers); Defect Disposition, (Defects can be True Defects or False Positives); and a prose Defect Description.

A small sample of a Defect Report appears in Figure 3.

2.3 Experiment Preparation

The participants were given a series of lectures on software requirements specifications, the SCR tabular requirements notation, inspection procedures, the defect classification scheme, and the filling out of data collection forms. The references for these lectures were Fagan [6], Parnas [13], and the IEEE Guide to Software Requirements Specifications [1].

The participants were then assembled into three-person teams – see Section 2.1.3 for details. Within each team, members were randomly assigned to act as the moderator, the recorder, or the reader during the collection meeting.

2.4 Conducting the Experiment

2.4.1 Training

For the training exercise, each team inspected the ELEVATOR SRS. Individual team members read the specification and recorded all defects they found on a Defect Report Form. Their efforts were restricted to two hours. Later we met with the participants and answered questions about the experimental procedures. Afterwards, each team conducted a supervised collection meeting and filled out a master Defect Report Form for the entire team. The ELEVATOR SRS was not used in the remainder of the experiment.

2.4.2 Experimental Phase

This phase involved two inspection rounds. The instruments used were the WLMS and CRUISE specifications discussed in Section 2.2.1, a checklist, three groups of defect-based scenarios, and the Defect Report Form. The development of the checklist and scenarios is described in Section 2.2.2. The same checklist and scenarios were used for both documents.

During the first Round, four of the eight teams were asked to inspect the CRUISE specification; the remaining four teams inspected the WLMS specification. The detection methods used by each team are shown in Table 1. Defect detection was limited to two hours, and all potential defects were reported on the Defect Report Form. After defect detection, all materials were collected.⁵

⁵For each round, we set aside 14 two-hour time slots during which inspection tasks could be done. Participants performed each task within a single two-hour session and were not allowed to work at other times.

Rev	Method	Sum	1	2			21	32			41	42
42	Data inconsistency	9	0	0			0	0			0	0
43	Incorrect functionality	6	0	1			0	0			0	0
44	Missing functionality	18	0	0	...		1	0	...		0	0
Team	Scenario	23	0	1			0	1			0	0
Key			AH	DT			MA	AH			DT	AH

Figure 4: Data Collection for each WLMS inspections. This figure shows the data collected from one team's WLMS inspection. The first three rows identify the review team members, the detection methods they used, the number of defects they found, and shows their individual defect summaries. The fourth row contains the team defect summary. The defect summaries show a 1 (0) where the team or individual found (did not find) a defect. The fifth row contains the defect key which identifies those reviewers who were responsible for the defect (AH for Ad Hoc only; CH for Checklist or Ad Hoc; DT for data type inconsistencies, Checklist, and Ad Hoc; IF for incorrect functionality, Checklist and Ad Hoc; and MA for missing or ambiguous functionality, Checklist and Ad Hoc). Meeting gain and loss rates can be calculated by comparing the individual and team defect summaries. For instance, defect 21 is an example of *meeting loss*. It was found by reviewer 44 during the defect detection activity, but the team did not report it at the collection meeting. Defect 32 is an example of *meeting gain*; it is first discovered at the collection meeting.

Once all team members had finished defect detection, the team's moderator arranged for the collection meeting. At the collection meeting, the documents were reread and defects discussed. The team's recorder maintained the team's master Defect Report Form. Collection was also limited to 2 hours. The entire Round was completed in one week.

The second Round was similar to the first except that teams who had inspected the WLMS during Round 1 inspected the CRUISE in Round 2 and vice versa.

3 Data and Analysis

3.1 Data

Three sets of data are important to our study: the defect key, the team defect summaries, and the individual defect summaries.

The defect key encodes which reviewers are responsible for each defect. In this study, reviewer responsibilities are defined by the detection techniques a reviewer uses. Ad Hoc reviewers are responsible (asked to search for) for all defects. Checklist reviewers are responsible for a large subset of the Ad Hoc defects⁶. Since each Scenario is a refinement of several Checklist items, each Scenario reviewer is responsible for a distinct subset of the Checklist

⁶i.e., defects for which an Ad Hoc reviewer is responsible.

Rev	Method	Sum	1	2	...	14	...	17	...	25	26
42	Ad Hoc	7	0	1		0		0		1	0
43	Ad Hoc	6	0	1		0		0		1	0
44	Ad Hoc	4	0	0		0		0		0	0
Team	Ad Hoc	10	0	1		1		0		1	0
Key			AH	MF		AH		AH		AH	DT

Figure 5: **Individual and Team Defect Summaries (CRUISE)**. This figure shows the data collected from one team's CRUISE inspection. The data is identical to that of the WLMS inspections except that the CRUISE has fewer defects – 26 versus 42 for the WLMS – and the defect key is different.

defects.

The team defect summary shows whether or not a team discovered a particular defect. This data is gathered from the defect report forms filled out at the collection meetings and is used to assess the effectiveness of each defect detection method.

The individual defect summary shows whether or not a reviewer discovered a particular defect. This data is gathered from the defect report forms each reviewer completed during their defect detection activity. Together with the defect key it is used to assess whether or not each detection technique improves the reviewer's ability to identify specific classes of defects.

We measure the value of collection meetings by comparing the team and individual defect summaries to determine the meeting gain and loss rates.

One team's individual and team defect summaries, and the defect key are represented in Figures 4 and Figure 5.

3.2 Analysis of Team Performance

Figure 6 summarizes the team performance data. As depicted, the Scenario detection method resulted in the highest defect detection rates, followed by the Ad Hoc detection method, and finally by Checklist the detection method.

Table 2 presents a statistical analysis of the team performance data as outlined in Section 2.1.5. The independent variables are listed from the most to the least significant. The Detection method and Specification are significant, but the Round, Replication, and Order are not.

Next, we analyzed the combined Instrumentation and Treatment effects. Table 3 shows the input to this

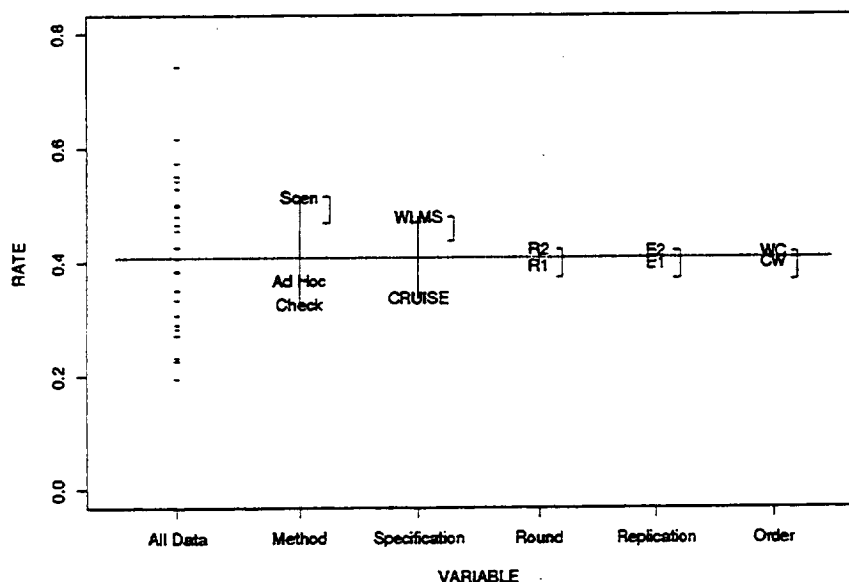


Figure 6: Defect Detection Rates by Independent Variable. The dashes in the far left column show each team's defect detection rate for the WLMS and CRUISE. The horizontal line is the average defect detection rate. The plot demonstrates the ability of each variable to explain variation in the defect detection rates. For the Specification variable, the vertical location of WLMS (CRUISE) is determined by averaging the defect detection rates for all teams inspecting WLMS (CRUISE). The vertical bracket,], to the right of each variable shows one standard error of the difference between two settings of the variable. The plot indicates that both the Method and Specification are significant; but Round, Replication, and Order are not.

Independent Variable	SS_T	ν_T	SS_R	ν_R	$(SS_T/\nu_T)(\nu_R/SS_R)$	Significance Level
Detection Method – treatment	.200	2	.359	29	8.064	< .01
Specification– instrumentation	.163	1	.396	30	12.338	< .01
Inspection round – maturation	.007	1	.551	30	.391	.54
Experimental run – replication	.007	1	.551	30	.391	.54
Order – presentation	.003	1	.003	30	.141	.71
Team composition – selection	.289	15	.268	16	1.151	.39

Table 2: Analysis of Variance for Each Independent Variable. The analysis of variance shows that only the choice of detection method and specification significantly explain variation in the defect detection rate. Team composition is also not significant.

analysis. Six of the cells contain the average detection rate for teams using each detection method and specification (3 detection methods applied to 2 specifications). The results of this analysis, shown in Table 4, indicate that the interaction between Specification and Method is not significant. This means that although the average detection rates varied for the two specifications, the effect of the detection methods is not linked to these differences. Therefore, we reject the null hypothesis that the detection methods have no effect on inspection performance.

Specification	Detection Method															
	Ad Hoc						Checklist			Scenario						
WLMS	.5	.38	.29	.5	.48	.45	.29	.52	.5	.33	.74	.57	.55	.4	.62	.55
(average)	.43						.41			.57						
Cruise	.46	.27	.27	.23	.38	.23	.35	.19	.31	.23	.23	.5	.42	.42	.54	.35
(average)	.31						.24			.45						

Table 3: Team Defect Detection Rate Data. The nominal and average defect detection rates for all 16 teams.

Effect	SS_T	ν_T	SS_R	ν_R	$(SS_T/\nu_T)(\nu_R/SS_R)$	Significance Level
Detection Method	.200	2	.212	26	12.235	< .01
Specification	.143	1	.212	26	17.556	< .01
Meth×Spec	.004	2	.212	26	.217	.806

Table 4: Analysis of Variance of Detection Method and Specification. This table displays the results of an analysis of the variance of the average detection rates given in Table 3.

3.3 Effect of Scenarios on Individual Performance

We initially hypothesized that increasing the specialization and coordination of each reviewer's responsibilities would improve team performance. We proposed that the Scenario would be one way to achieve this. We have shown above that the teams using Scenarios were the most effective. However, this did not establish that the improvement was due to increases in specialization and coordination, and not to some other factor.

Consequently, our concern is to determine exactly how the use of Scenarios affected the reviewer's performance. To examine this, we formulated two hypothesis schemas.

- H1: Method X reviewers do not find any more X defects than do method Y reviewers.
- H2: Method X reviewers find either a greater or smaller number of non X defects than do method Y reviewers.

Alternative explanations for the observed improvement could be (1) the Scenario reviewers responded to some perceived expectation that their performance should improve; or (2) the Scenario approach improves individual performance regardless of Scenario content.

3.3.1 Rejecting the Perceived Expectation Argument

If Scenario reviewers performed better than Checklist and Ad Hoc reviewers on both scenario-targeted and non-scenario-targeted defects, then we must consider the possibility that their improvement was caused by something

Reviewers Using Method		Finding Defects of Type		Compared with Reviewers using Method				
Detection Method	Number Reviewers	Defect Population	Number Present	DT	MF	IF	CH	AH
DT	6	DT	14	- (6.5)	.02 (3)	.06 (4.5)	.01 (4)	.02 (4)
MF	6	MF	5	.07 (0.5)	- (2)	.12 (1)	.02 (0)	.04 (1)
IF	6	IF	5	.01 (0)	.01 (1)	- (1.5)	.04 (1)	.01 (1)
CH	12	CH	38	.95 (10.5)	.86 (11)	.89 (12.5)	- (8)	.51 (10)
AH	18	AH	42	.91 (12)	.84 (12.5)	.75 (13)	.37 (9.5)	- (11)

Table 5: **Significance Table for H1 hypotheses: WLMS inspections.** This table tests the H1 hypothesis - Method X reviewers do not find any more X defects than do method Y reviewers - for all pairs of detection methods. Each row in the table corresponds to a population of reviewers and the population of defects for which they were responsible, i.e., method X reviewers and X defects. The last five columns correspond to a second reviewer population, i.e., method Y reviewers. Each cell in the last five columns contains two values. The first value is the probability that H1 is true, using the one-sided Wilcoxon-Mann-Whitney test. The second value - in parentheses - is the median number of defects found by the method Y reviewers.

Reviewers Using Method		Finding Defects of Type		Compared with Reviewers using Method				
Detection Method	Number Reviewers	Defect Population	Number Present	DT	MF	IF	CH	AH
DT	5	DT	10	- (6)	.05 (3)	.03 (2)	< .01 (1)	.02 (3)
MF	5	MF	1	NA (0)	- (0)	NA (0)	NA (0)	NA (0)
IF	5	IF	3	NA (0)	NA (0)	- (0)	NA (0)	NA (0)
CH	12	CH	24	> .99 (8)	.95 (5)	.93 (4)	- (2.5)	.98 (5)
AH	21	AH	26	.96 (8)	.50 (5)	.41 (5)	.02 (3)	- (5)

Table 6: **Significance Table for H1 hypotheses: CRUISE inspections.** This analysis is identical to that performed for WLMS inspections. However, we chose not to perform any statistical analysis for the Missing Functionality and Incorrect Functionality defects because there are too few defects of those types.

Reviewers Using Method		Finding Defects of Type		Compared with Reviewers using Method				
Detection Method	Number Reviewers	Defect Population	Number Present	DT	MF	IF	CH	AH
DT	6	DT ^c	28	- (4.5)	.92 (9)	.82 (7.5)	.50 (5.5)	.64 (6)
MF	6	MF ^c	37	.87 (11)	- (9.5)	.83 (12.5)	.56 (8.5)	.64 (10)
IF	6	IF ^c	37	.66 (11)	.53 (12)	- (11.5)	.24 (8.5)	.27 (10)
CH	12	CH ^c	4	.12 (0.5)	.28 (1)	.35 (1)	- (1)	.07 (1)
AH	18	AH ^c	0	NA (0)	NA (0)	NA (0)	NA (0)	- (0)

Table 7: **Significance Table for H2 hypothesis: WLMS inspections.** This table tests the H2 hypothesis - Method X reviewers find a greater or smaller number of non X defects than do method Y reviewers - for all pairs of detection methods. Each row in the table corresponds to a population of reviewers and the population of defects for which they were not responsible - i.e., method X reviewers and non X defects (the complement of the set of X defects). The last five columns correspond to a second reviewer population, i.e., method Y reviewers. Each cell in the last five columns contains two values. The first value is the probability that H2 is true, using the two-sided Wilcoxon-Mann-Whitney test. The second value is the median number of defects found by the method Y reviewers.

other than the scenarios themselves.

One possibility was that the Scenario reviewers were merely reacting to the novelty of using a clearly different approach, or to a perceived expectation on our part that their performance should improve. To examine this we analyzed the individual defect summaries to see how Scenario reviewers differed from other reviewers.

The detection rates of Scenario reviewers⁷ are compared with those of all other reviewers in Tables 5, 6, 7 and 8. Using the one and two-sided Wilcoxon-Mann-Whitney tests [15], we found that in most cases Scenario reviewers were more effective than Checklist or Ad Hoc reviewers at finding the defects the scenario was designed to uncover. At the same time, all reviewers, regardless of which detection method each used, were equally effective at finding those defects not targeted by any of the Scenarios.

Since Scenario reviewers could not have known the defect classifications, it is unlikely that their reporting could have been biased. Therefore these results suggest that the detection rate of Scenario reviewers shows improvement only with regard to those defects for which they are explicitly responsible. Consequently, the argument that the Scenario reviewers' improved performance was primarily due to raised expectations or unknown motivational factors is not supported by the data.

⁷i.e., reviewers using Scenarios.

Reviewers Using Method		Finding Defects of Type		Compared with Reviewers using Method				
Detection Method	Number Reviewers	Defect Population	Number Present	DT	MF	IF	CH	AH
DT	5	DT ^c	16	- (2)	.59 (2)	.86 (3)	.37 (2)	.46 (2)
MF	5	MF ^c	25	.96 (8)	- (5)	.33 (4)	.06 (3)	.62 (5)
IF	5	IF ^c	23	.96 (8)	.41 (4)	- (5)	.44 (2.5)	.57 (5)
CH	12	CH ^c	2	NA (0)	NA (1)	NA (0)	- (0)	NA (0)
AH	21	AH ^c	0	NA (0)	NA (0)	NA (0)	NA (0)	- (0)

Table 8: **Significance Table for H2 hypothesis: CRUISE inspections.** This analysis is identical to that performed for WLMS inspections. However, we chose not to perform statistical analysis for the non non Checklist defects because there are too few defects of that type.

3.3.2 Rejecting the General Improvement Argument

Another possibility is that the Scenario approach rather than the content of the Scenarios was responsible for the improvement.

Each Scenario targets a specific set of defects. If the reviewers using a type X Scenario had been no more effective at finding type X defects than had reviewers using non-X Scenarios, then the content of the Scenarios did not significantly influence reviewer performance. If the reviewers using a type X Scenario had been more effective at finding non-X defects than had reviewers using other Scenarios, then some factor beyond content caused the improvement.

To explore these possibilities we compared the Scenario reviewers' individual defect summaries with each other.

Looking again at Tables 5, 6, 7, and 8 we see that each group of Scenario reviewers were the most effective at finding the defects their scenarios were designed to detect, but were generally no more effective than other Scenario reviewers at finding defects their Scenarios were not designed to detect.

Since Scenario reviewers showed improvement only in finding the defects for which they were explicitly responsible, we conclude that the content of the Scenario was primarily responsible for the improved reviewer performance.

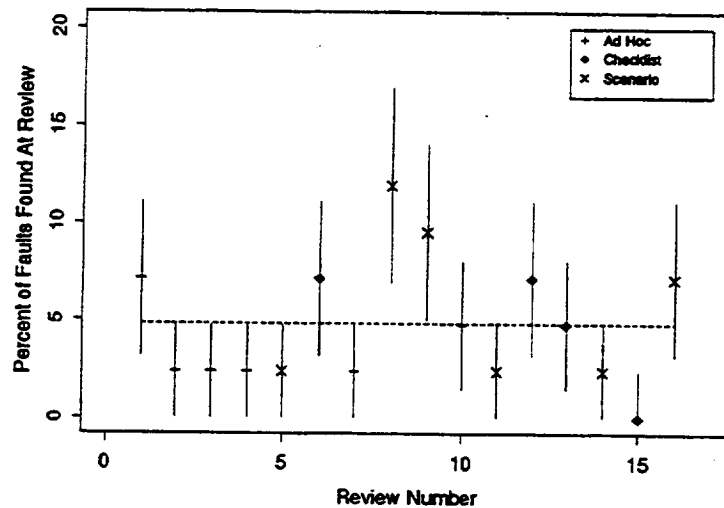


Figure 7: Meeting Gains for WLMS Inspections. Each point represents the meeting gain rate for a single inspection, i.e., the number of defects first identified at a collection meeting divided by the total number of defects in the specification. Each rate is marked with symbol indicating the inspection method used. The vertical line segment through each symbol indicates one standard deviation in the estimate (assuming each defect was a Bernoulli trial). This information helps in assessing the significance of any one rate. The average meeting gain rate is $4.7 \pm 1.3\%$ for the WLMS. ($3.1 \pm 1.1\%$ for the CRUISE.)

3.4 Analysis of Checklists on Individual Performance

The scenarios used in this study were derived from the checklist. Although this checklist targeted a large number of existing defects, our analysis shows that the performance of Checklist teams were no more effective than Ad Hoc teams. One explanation for this is that nonsystematic techniques are difficult for reviewers to implement.

To study this explanation we again tested the H1 hypothesis that Checklist reviewers were no more effective than Ad Hoc reviewers at finding Checklist defects.

From Tables 5 and 6 we see that even though the Checklist targets a large number of defects, it does not actually improve a reviewer's ability to find those defects.

3.5 Analysis of Collection Meetings

In his original paper on software inspections Fagan [6] asserts that

Sometimes flagrant errors are found during ... [defect detection], but in general, the number of errors found is not nearly as high as in the ... [collection meeting] operation.

From a study of over 50 inspections, Votta [17] collected data that strongly contradicts this assertion. In this Section, we measure the benefits of collection meetings by comparing the team and individual defect summaries to determine the meeting gain and meeting loss rates. (See Figure 4 and Figure 5).

A "meeting gain" occurs when a defect is found for the first time at the collection meeting. A "meeting loss" occurs when a defect is first found during an individual's defect detection activity, but it is subsequently not recorded during the collection meeting. Meeting gains may thus be offset by meeting losses and the difference between meeting gains and meeting losses is the net improvement due to collection meetings.

Our results indicate that collection meetings produce no net improvement.

3.5.1 Meeting Gains

The meeting gain rates reported by Votta were a negligible $3.9 \pm .7\%$. Our data tells a similar story. (Figure 7 displays the meeting gain rates for WLMS inspections.) The mean gain rate is $4.7 \pm 1.3\%$ for WLMS inspections and $3.1 \pm 1.1\%$ for CRUISE inspections. The rates are not significantly different.

It is interesting to note that these results are consistent with Votta's earlier study even though Votta's reviewers were professional software developers and not students.

3.5.2 Meeting Losses

The average meeting loss rates were $6.8 \pm 1.6\%$ and $7.7 \pm 1.7\%$ for the WLMS and CRUISE respectively. (See Figure 8.)

One cause of meeting loss might be that reviewers are talked out of the belief that something is a defect. Another cause may be that during the meeting reviewers forget or can not reconstruct a defect found earlier.

This effect has not been previously reported in the literature. However, since the interval between the detection and collection activities is usually longer in practice than it was in our experiment (one to two days in our study versus one or two weeks in practice), this effect may be quite significant.

3.5.3 Net Meeting Improvement

The average net meeting improvement is -0.9 ± 2.2 for WLMS inspections and -1.2 ± 1.7 for CRUISE inspections. (Figure 9 displays the net meeting improvement for WLMS inspections.) We found no correlations between the loss, gain, or net improvement rates and any of our experiment's independent variables.

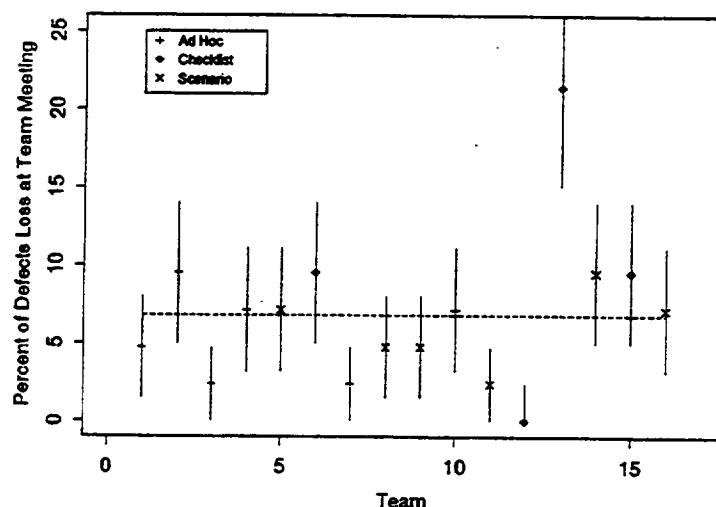


Figure 8: Meeting Loss Rate for WLMS Inspections. Each point represents the meeting loss rate for a single inspection. The meeting loss rate is the number of defects first detected by an individual reviewer divided by the total number of defects in the specification. Each rate is marked with a symbol indicating the inspection method used. The vertical line segment through each symbol indicates one standard deviation in the estimate of the rate (assuming each fault was a Bernoulli trial). This information helps in determining the significance of any one rate. The average team loss rate is $6.8 \pm 1.6\%$ for the WLMS. ($7.7 \pm 1.7\%$ for CRUISE).

4 Summary and Conclusions

Our experimental design for comparing defect detection methods is flexible and economical, and allows the experimenter to assess several potential threats to the experiment's internal validity. In particular, we determined that neither maturation, replication, selection, or presentation effects had any significant influence on inspection performance. However, differences in the SRS did.

From our analysis of the experimental data we drew several conclusions.

1. The defect detection rate when using Scenarios is superior to that obtained with Ad Hoc or Checklist methods – an improvement of roughly 35%.
2. Scenarios help reviewers focus on specific defect classes. Furthermore, in comparison to Ad Hoc or Checklist methods, their ability to detect other classes of defects is not compromised. (It should be noted however, that the scenarios appeared to be better suited to the defect profile of the WLMS than the CRUISE. This indicates that poorly designed scenarios may lead to poor inspection performance.)
3. The Checklist method – the industry standard, was no more effective than the Ad Hoc

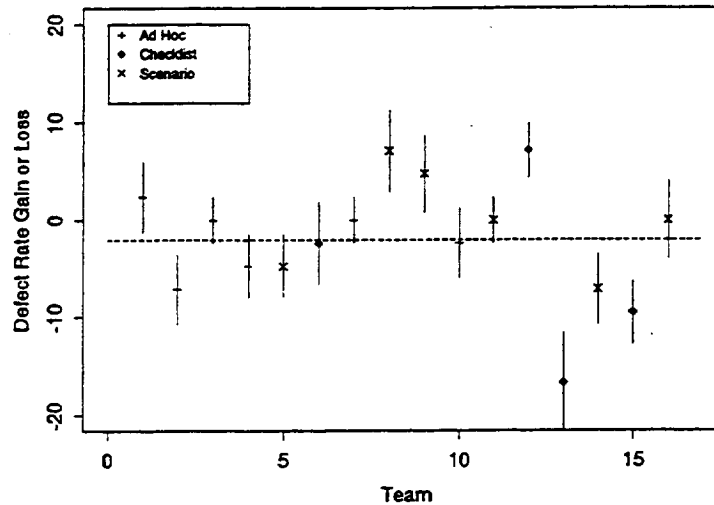


Figure 9: Net Meeting Improvement for WLMS. Each symbol indicates the net meeting improvement for a single inspection. The average net meeting improvement rate is -0.9 ± 2.2 for the WLMS. (-1.2 ± 1.7 for the CRUISE). These rates are not significantly different from 0.

detection method.

4. On the average, collection meetings contribute nothing to defect detection effectiveness.

The results of this work have important implications for software practitioners. The indications are that overall inspection performance can be improved when individual reviewers use systematic procedures to address a small set of specific issues. This contrasts with the usual practice, in which reviewers have neither systematic procedures nor clearly defined responsibilities.

Economical experimental designs are necessary to allow replication in other environments with different populations. For software researchers, this work demonstrates the feasibility of constructing and executing inexpensive experiments to validate fundamental research recommendations.

5 Future Work

The experimental data raise many interesting questions for future study.

- In many instances a single reviewer found a defect, but the defect was not subsequently recorded at the collection meeting. Are single reviewers sometimes forgetting to mention defects they observed, or is

the reviewer being talked out of the defect at the team meeting? What are the significant suppression mechanisms affecting collection meetings?

- Very few defects are initially discovered during collection meetings. Therefore, in view of their impact on production interval, are these meetings worth holding?
- More than half of the defects are not addressed by the Scenarios used in this study. What other Scenarios are necessary to achieve a broader defect coverage?
- There are several threats to this experiment's external validity. These threats can only be addressed by replicating and reproducing these studies. Each new run reduces the probability that our results can be explained by human variation or experimental error. Consequently, we are creating a laboratory kit (i.e., a package containing all the experimental materials, data, and analysis) to facilitate replication. The kit should be publicly available by June, 1994.
- Finally, we are using the lab kit to reproduce the experiments with other university researchers in Japan, Germany, Italy, and Australia and with industrial developers at AT&T Bell Laboratories and Motorola Inc. These studies will allow us to evaluate our hypotheses with different populations of programmers and different software artifacts.

Acknowledgments

We would like to recognize the efforts of the experimental participants – an excellent job was done by all. Our thanks to Mark Ardis, John Kelly, and David Weiss, who helped us to identify sample requirements specifications and inspection checklists, and to John Gannon, Richard Gerber, Clive Loader, Eric Slud and Scott VanderWeil for their valuable technical comments. Finally, Art Caso's editing is greatly appreciated.

References

- [1] *IEEE Guide to Software Requirements Specifications*. Soft. Eng. Tech. Comm. of the IEEE Computer Society, 1984. IEEE Std 830-1984.
- [2] V. R. Basili and D. M. Weiss. Evaluation of a software requirements document by analysis of change data. In *Proceedings of the Fifth International Conference on Software Engineering*, pages 314-323, San Diego, CA, March 1981.
- [3] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall, Englewood Cliffs, NJ, 1981.
- [4] G. E. P. Box, W. G. Hunter, and J. S. Hunter. *Statistics for Experimenters*. John Wiley & Sons, New York, 1978.
- [5] Stephen G. Eick, Clive R. Loader, M. David Long, Scott A. Vander Wiel, and Lawrence G. Votta. Estimating software fault content before coding. In *Proceedings of the 14th International Conference on Software Engineering*, pages 59-65, May 1992.
- [6] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182-211, 1976.
- [7] Kathryn L. Heninger. Specifying Software Requirements for Complex Systems: New Techniques and their Application. *IEEE Transactions on Software Engineering*, SE-6(1):2-13, January 1980.
- [8] Watts S. Humphrey. *Managing the Software Process*. Addison-Wesley Publishing Co., 1989. Reading, Massachusetts.
- [9] *IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software*. Soft. Eng. Tech. Comm. of the IEEE Computer Society, 1989. IEEE Std 982.2-1988.
- [10] *IEEE Standard for software reviews and audits*. Soft. Eng. Tech. Comm. of the IEEE Computer Society, 1989. IEEE Std 1028-1988.
- [11] Charles M. Judd, Eliot R. Smith, and Louise H. Kidder. *Research Methods in Social Relations*. Holt, Rinehart and Winston, Inc., Fort Worth, TX, sixth edition, 1991.
- [12] J. Kirby. Example NRL/SCR software requirements for an automobile cruise control and monitoring system. Technical Report TR-87-07, Wang Institute of Graduate Studies, July 1984.
- [13] Dave L. Parnas and David M. Weiss. Active design reviews: principles and practices. In *Proceedings of the 8th International Conference on Software Engineering*, pages 215-222, Aug. 1985.
- [14] G. Michael Schnieder, Johnny Martin, and W. T. Tsai. An experimental study of fault detection in user requirements. *ACM Transactions on Software Engineering and Methodology*, 1(2):188-204, April 1992.
- [15] S. Siegel and N.J. Castellan, Jr. *Nonparametric Statistics For the Behavioral Sciences*. McGraw-Hill Book Company, New York, NY, second edition, 1988.
- [16] J. vanSchouwen. The A-7 requirements model: Re-examination for real-time systems and an application to monitoring systems. Technical Report TR-90-276, Queen's University, Kingston, Ontario, Canada, May 1990.
- [17] Lawrence G. Votta. Does every inspection need a meeting? In *Proceedings of ACM SIGSOFT '93 Symposium on Foundations of Software Engineering*. Association for Computing Machinery, December 1993.
- [18] William G. Wood. Temporal logic case study. Technical Report CMU/SEI-89-TR-24, Software Engineering Institute, Pittsburgh, PA, August 1989.

A Ad Hoc Detection

The defect taxonomy is due to the work of Schneider, et al., and Basili and Weiss.

- Omission

- Missing Functionality: Information describing the desired internal operational behavior of the system has been omitted from the SRS.
- Missing Performance: Information describing the desired performance specifications has either been omitted or described in a way that is unacceptable for acceptance testing.
- Missing Interface: Information describing how the proposed system will interface and communicate with objects outside the the scope of the system has been omitted from the SRS.
- Missing Environment: Information describing the required hardware, software, database, or personnel environment in which the system will run has been omitted from the SRS

- Commission

- Ambiguous Information: An important term, phrase or sentence essential to the understanding of system behavior has either been left undefined or defined in a way that can cause confusion and misunderstanding.
- Inconsistent Information: Two sentences contained in the SRS directly contradict each other or express actions that cannot both be correct or cannot both be carried out.
- Incorrect Fact: Some sentence contained in the SRS asserts a facts that cannot be true under the conditions specified in the SRS.
- Wrong Section: Essential information is misplaced within the SRS

B Checklist Method

- General

- Are the goals of the system defined?
- Are the requirements clear and unambiguous?
- Is a functional overview of the system provided?
- Is an overview of the operational modes provided?
- Have the software and hardware environments been specified?
- If assumptions that affect implementation have been made, are they stated?
- Have the requirements been stated in terms of inputs, outputs, and processing for each function?
- Are all functions, devices, constraints traced to requirements and vice versa?
- Are the required attributes, assumptions and constraints of the system completely listed?

- Omission

- Missing Functionality
 - * Are the described functions sufficient to meet the system objectives?
 - * Are all inputs to a function sufficient to perform the required function?
 - * Are undesired events considered and their required responses specified?
 - * Are the initial and special states considered (e.g., system initiation, abnormal termination)?
- Missing Performance
 - * Can the system be tested, demonstrated, analyzed, or inspected to show that it satisfies the requirements?
 - * Have the data type, rate, units, accuracy, resolution, limits, range and critical values for all internal data items been specified?
 - * Have the accuracy, precision, range, type, rate, units, frequency, and volume of inputs and outputs been specified for each function?
- Missing Interface
 - * Are the inputs and outputs for all interfaces sufficient?
 - * Are the interface requirements between hardware, software, personnel, and procedures included?
- Missing Environment
 - * Have the functionality of hardware or software interacting with the system been properly specified?

- Commission

- Ambiguous Information
 - * Are the individual requirements stated so that they are discrete, unambiguous, and testable?
 - * Are all mode transitions specified deterministically?
- Inconsistent Information
 - * Are the requirements mutually consistent?
 - * Are the functional requirements consistent with the overview?
 - * Are the functional requirements consistent with the actual operating environment?
- Incorrect or Extra Functionality
 - * Are all the described functions necessary to meet the system objectives?
 - * Are all inputs to a function necessary to perform the required function?
 - * Are the inputs and outputs for all interfaces necessary?
 - * Are all the outputs produced by a function used by another function or transferred across an external interface?
- Wrong Section
 - * Are all the requirements, interfaces, constraints, etc. listed in the appropriate sections.

C Scenarios

C.1 Data Type Consistency Scenario

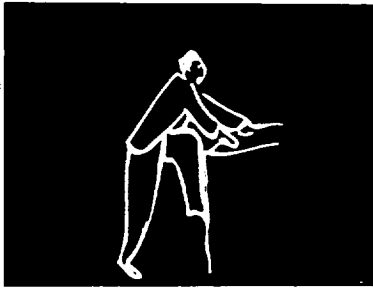
1. Identify all data objects mentioned in the overview (e.g., hardware component, application variable, abbreviated term or function)
 - (a) Are all data objects mentioned in the overview listed in the external interface section?
2. For each data object appearing in the external interface section determine the following information:
 - Object name:
 - Class: (e.g., input port, output port, application variable, abbreviated term, function)
 - Data type: (e.g., integer, time, boolean, enumeration)
 - Acceptable values: Are there any constraints, ranges, limits for the values of this object
 - Failure value: Does the object have a special failure value?
 - Units or rates:
 - Initial value:
 - (a) Is the object's specification consistent with its description in the overview?
 - (b) If object represents a physical quantity, are its units properly specified?
 - (c) If the object's value is computed, can that computation generate a non-acceptable value?
3. For each functional requirement identify all data object references:
 - (a) Do all data object references obey formatting conventions?
 - (b) Are all data objects referenced in this requirement listed in the input or output sections?
 - (c) Can any data object use be inconsistent with the data object's type, acceptable values, failure value, etc.?
 - (d) Can any data object definition be inconsistent with the data object's type, acceptable values, failure value, etc.?

C.2 Incorrect Functionality Scenario

1. For each functional requirement identify all input/output data objects:
 - (a) Are all values written to each output data object consistent with its intended function?
 - (b) Identify at least one function that uses each output data object.
2. For each functional requirement identify all specified system events:
 - (a) Is the specification of these events consistent with their intended interpretation?
3. Develop an invariant for each system mode (i.e. Under what conditions must the system exit or remain in a given mode)?
 - (a) Can the system's initial conditions fail to satisfy the initial mode's invariant?
 - (b) Identify a sequence of events that allows the system to enter a mode without satisfying the mode's invariant.
 - (c) Identify a sequence of events that allows the system to enter a mode, but never leave (deadlock).

C.3 Ambiguities Or Missing Functionality Scenario

1. Identify the required precision, response time, etc. for each functional requirement.
 - (a) Are all required precisions indicated?
2. For each requirement, identify all monitored events.
 - (a) Does a sequence of events exist for which multiple output values can be computed?
 - (b) Does a sequence of events exist for which no output value will be computed?
3. For each system mode, identify all monitored events.
 - (a) Does a sequence of events exist for which transitions into two or more system modes is allowed?



Software Process Evolution at the SEL

VICTOR BASILI, *University of Maryland*
SCOTT GREEN, *NASA Goddard Space Flight Center*

◆ *The Software Engineering Laboratory has been adapting, analyzing, and evolving software processes for the last 18 years. Their approach is based on the Quality Improvement Paradigm, which is used to evaluate process effects on both product and people. The authors explain this approach as it was applied to reduce defects in code.*

Since 1976, the Software Engineering Laboratory of the National Aeronautics and Space Administration's Goddard Space Flight Center has been engaged in a program of understanding, assessing, and packaging software experience. Topics of study include process, product, resource, and defect models, as well as specific technologies and tools. The approach of the SEL — a consortium of the Software Engineering Branch of NASA Goddard's Flight Dynamics Division, the Computer Science Department of the University of Maryland, and the Software Engineering Operation of Computer Sciences Corp. — has been to gain an in-depth understanding of project and environment characteristics using process models and baselines. A process is evaluated for study, applied experimentally to a project, analyzed with respect to baselines and process model, and evaluated in terms of the experiment's goals. Then on the basis of the experiment's conclusions, results are packaged and the process is tailored for improvement, applied again, and reevaluated.

In this article, we describe our improvement approach, the Quality Improvement Paradigm, as the SEL applied it to reduce code defects by emphasizing reading techniques. The box on p. 63 describes the Quality Improvement Paradigm in detail. In examining and adapting reading techniques, we go through a systematic process of evaluating the candidate

process and refining its implementation through lessons learned from previous experiments and studies.

As a result of this continuous, evolutionary process, we determined that we could successfully apply key elements of the Cleanroom development method in the SEL environment, especially for projects involving fewer than 50,000 lines of code (all references to lines of code refer to developed, not delivered, lines of code). We saw indications of lower error rates, higher productivity, a more complete and consistent set of code comments, and a redistribution of developer effort. Although we have not seen similar reliability and cost gains for larger efforts, we continue to investigate the Cleanroom method's effect on them.

EVALUATING CANDIDATE PROCESSES

To enhance the possibility of improvement in a particular environment, the SEL introduces and evaluates new technology within that environment. This involves experimentation with the new technology, recording findings in the context of lessons learned, and adjusting the associated processes on the basis of this experience. When the technology is notably risky — substantially different from what is familiar to the environment — or requires more detailed evaluation than would normally be expended, the SEL conducts experimentation off-line from the project environment.

Off-line experiments may take the form of either controlled experiments or case studies. Controlled experiments are warranted when the SEL needs a detailed analysis with statistical assurance in the results. One problem with controlled experiments is that the project must be small enough to replicate the experiment several times. The SEL then performs a case study to validate the results on a project of credible size that is representative of the environment. The case study adds

validity and credibility through the use of typical development systems and professional staff. In analyzing both controlled experiments and case studies, the Goal/Question/Metric paradigm, described in the box on p. 63, provides an important framework for focusing the analysis.

On the basis of experimental results, the SEL packages a set of lessons learned and makes them available in an experience base for future analysis and application of the technology.

Experiment 1: Reading versus testing.

Although the SEL had historically been a test-driven organization, we decided to experiment with introducing reading techniques. We were particularly interested in how reading would compare with testing for fault detection. The goals of the first off-line, controlled experiment¹ were to analyze and compare code reading, functional testing, and structural testing, and to evaluate them with respect to fault-detection effectiveness, cost, and classes of faults detected.

We needed an analysis from the viewpoint of quality assurance as well as a comparison of performance with respect to software type and programmer experience. Using the GQM paradigm, we generated specific questions on the basis of these goals.

We had subjects use reading by stepwise abstraction,² equivalence-partitioning boundary-value testing, and statement-coverage structural testing.

We conducted the experiment twice at the University of Maryland on graduate students (42 subjects) and once at NASA Goddard (32 subjects). The experiment structure was a fractional factorial design, in which every subject applied each technique on a different program. The programs included a text formatter, a plotter, an abstract data type, and a database, and they ranged from 145 to 365 lines of code. We seeded each program with faults. The reading performed was at the unit level.

Although the results from both experiments support the emphasis on reading techniques, we report only the results of the controlled experiment on the NASA Goddard subjects because it involved professional developers in the target environment.

Figure 1 shows the fault-detection effectiveness and rate for each approach for the NASA Goddard experiment. Reading by stepwise abstraction proved superior to testing

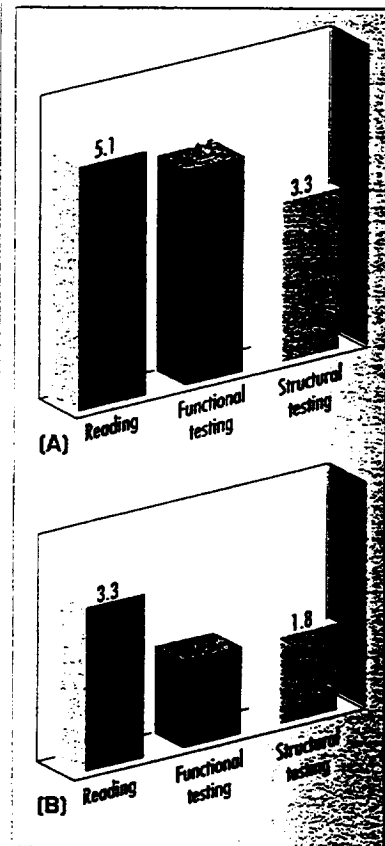
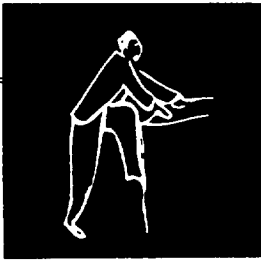


Figure 1. Results of the reading-versus-testing controlled experiment, in which reading was compared with functional and structural testing. (A) Mean number of faults detected for each technique and (B) number of faults detected per hour of use for each technique.



techniques in both the effectiveness and cost of fault detection, while obviously using fewer computer resources.

Even more interesting was that the subjects did a better job of estimating the code quality using reading than they did using testing. Readers thought they had found only about half the faults (which was nominally correct), while functional testers felt that had found essentially all the faults (which was never correct).

Furthermore, after completing the experiment, more than 90 percent of the participants thought functional testing had been the most effective technique, although the results clearly showed otherwise. This gave us some insight into the psychological effects of reading versus testing. Perhaps one reason testing appeared more satisfying was that the successful execution of multiple test cases generated a greater comfort level with the product quality, actually providing the tester with a false sense of confidence.

Reading was also more effective in uncovering most classes of faults, including interface faults. This told us

that perhaps reading might scale up well on larger projects.

Experiment 2: Validation with Cleanroom.

On the basis of these results, we decided to emphasize reading techniques in the SEL environment. However, we saw little improvement in overall reliability of the development systems. Part of the reason may have been that SEL project personnel had developed such faith in testing that the quality of their reading was relaxed, with the assumption that testing would ultimately uncover the same faults. We conducted a small off-line experiment at the University of Maryland to test this hypothesis; the results supported our assumption. (We did this on a small scale just to verify our hypothesis before continuing with the Cleanroom experiment.)

Why the Cleanroom method? The Cleanroom method emphasizes human discipline in the development process, using a mathematically based design approach and a statistical testing approach based on anticipated opera-

tional use.³ Development and testing teams are independent, and all development-team activities are performed without on-line testing.

Techniques associated with the method are the use of box structures and state machines, reading by stepwise abstraction, formal correctness demonstrations, and peer review. System development is performed through a pipeline of small increments to enhance concentration and permit testing and development to occur in parallel.

Because the Cleanroom method removes developer testing and relies on human discipline, we felt it would overcome the psychological barrier of reliance on testing.

Applying the QIP. The first step of the Quality Improvement Paradigm is to characterize the project and its environment. The removal of developer unit testing made the Cleanroom method a high-risk technology. Again, we used off-line experimentation at the University of Maryland as a mitigating approach.⁴ The environment was a laboratory course at the university, and the project involved an electronic message system of about 1,500 LOC. The experiment structure was a simple replicated design, in which control and experiment teams are defined. We assigned 10 three-person experiment teams to use the Cleanroom method. We gave five three-person control teams the same development methodology, but allowed them to test their systems. Each team was allowed five independent test submissions of their programs. We collected data on programmer background and attitude, computer-resource activity, and actual testing results.

The second step in the Quality Improvement Paradigm is to set goals. The goal here was to analyze the effects of the Cleanroom approach and evaluate it with respect to process, product, and participants, as compared with the non-Cleanroom approach.

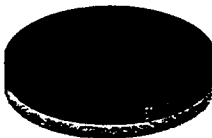
	Sample measures	Sample baselines	Sample expectations
Process	Effort distribution Change profile		Increased design effort because of emphasis on peer-review process
Cost	Productivity Level of rework Impact of specification changes	Historically, 26 lines of code per day	No degradation from current level
Reliability	Error rate Error distribution Error source	Historically, seven errors per thousand lines of code	Decreased error rate

Figure 2. Sample measures, baselines, and expectations for the case studies investigating the Cleanroom method.

We generated questions corresponding to this goal, focusing on the method's effect on each aspect being studied.

The next step of the Quality Improvement Paradigm involves selecting an appropriate process model. The process model selected for this experiment was the Cleanroom approach as defined by Harlan Mills at IBM's Federal Systems Division, but modified for our environment. For example, the graduate-student assistant for the course served as each group's independent test team. Also, because we used a language unfamiliar to the subjects to prevent bias, there was a risk of errors due solely to ignorance about the language. We therefore allowed teams to cleanly compile their code before submitting it to the tester.

Because of the nature of controlled experimentation, we made few modifications during the experiment.

Cleanroom's effect on the software-development process resulted in the Cleanroom developers more effectively applying the off-line reading techniques; the non-Cleanroom teams focused their efforts more on functional testing than reading. The Cleanroom teams spent less time on-line and were more successful in making scheduled deliveries. Further analysis revealed that the Cleanroom products had less dense complexity, a higher percentage of assignment statements, more global data, and more code comments. These products also more completely met the system requirements and had a higher percentage of successful independent test cases.

The Cleanroom developers indicated that they modified their normal software-development activities by doing a more effective job of reading, though they missed the satisfaction of actual program execution. Almost all said they would be willing to use Cleanroom on another development assignment.

Through observation, it was also clear that the Cleanroom developers

did not apply the formal methods associated with Cleanroom very rigorously. Furthermore, we did not have enough failure data or experience with Cleanroom testing to apply a reliability model. However, general analysis did indicate that the Cleanroom approach had potential payoff, and that additional investigation was warranted.

You can also view this experiment from the following perspective: We applied two development approaches. The only real difference between them was that the control teams had one extra piece of technology (developer testing), yet they did not perform as well as the experiment teams. One explanation might be that the control group did not use the available nontesting techniques as effectively because they knew they could rely on testing to detect faults. This supports our earlier findings associated with the reading-versus-testing experiment.

EVOLVING SELECTED PROCESS

The positive results gathered from these two experiments gave us the justification we needed to explore the Cleanroom method in case studies, using typical development systems as data points. We conducted two case studies to examine the method, again following the steps of the Quality Improvement Paradigm. A third case study was also recently begun.

First case study. The project we selected, Project 1, involved two subsystems from a typical attitude ground-support system. The system performs ground processing to determine a spacecraft's attitude, receiving and processing spacecraft telemetry data to meet the requirements of a particular mission.

The subsystems we chose are an integral part of attitude determination and are highly algorithmic. Both are interactive programs that together contain approximately 40,000 LOC, representing about 12

percent of the entire attitude ground-support system. The rest of the ground-support system was developed using the standard SEL development methodology.

The project was staffed principally by five people from the Flight Dynamics Division, which houses the SEL. All five were also working on other projects, so only part of their time

was allocated to the two subsystems. Their other responsibilities often took time and attention away from the case study, but this partial allocation represents typical staffing in this environment. All other projects with which the Project 1 staff were involved were non-Cleanroom efforts, so staff members would often be required to use multiple development methodologies during the same workday.

The primary goal of the first case study was to increase software quality and reliability without increasing cost. We also wanted to compare the characteristics of the Cleanroom method with those typical of the FDD environment. A well-calibrated baseline was available for comparison that described a variety of process characteristics, including effort distribution, change rates, error rates, and productivity. The baseline represents the history of many earlier SEL studies. Figure 2 shows a sample of the expected variations from the SEL baselines for a set of process characteristics.

Choosing and tailoring processes. The process models available for examination were the standard SEL model,⁵ which represents a reuse-oriented waterfall life-cycle model; the

**ALMOST
ALL THE
CLEANROOM
TEAM SAID
THEY'D USE
THE METHOD
AGAIN.**



IBM/FSD Cleanroom model, which appeared in the literature and was available through training; and the experimental University of Maryland Cleanroom model, which was used in the earlier controlled experiment.⁴

We examined the lessons learned from applying the IBM and University of Maryland models. The results from the IBM model were notably positive, showing that the basic process, methods, and techniques were effective for that particular environment. However, the process model had been applied by the actual developers of the methodology, in the environment for which it was developed. The University of Maryland model also had specific lessons, including the effects of not allowing developers to test their code, the effectiveness of the process on a small project, and the conclusion that formal methods appeared particularly difficult to apply and required specific skills.

On the basis of these lessons and the characteristics of our environment, we selected a Cleanroom process model with four key elements:

- ◆ separation of development and test teams,
- ◆ reliance on peer review instead of unit-level testing as the primary developer verification technique,
- ◆ use of informal state machines and functions to define the system design, and

- ◆ a statistical approach to testing based on operational scenarios.

We also provided training for the subjects, consistent with a University of Maryland course on the Cleanroom process model, methods, and techniques, with emphasis on reading through stepwise abstraction. We also stressed code reading by multiple reviewers because stepwise abstraction was new to many subjects. Michael Dyer and Terry Baker of IBM/FSD

provided additional training and motivation by describing IBM's use of Cleanroom.

To mitigate risk and address the developers' concerns, we examined backout options for the experiment. For example, because the subsystems were highly mathematical, we were afraid it would be difficult to find and correct mathematical errors without any developer testing. Because the project was part of an operational system with mission deadlines, we discussed options that ranged from allowing developer unit testing to discontinuing Cleanroom altogether. These discussions helped allay the primary apprehension of NASA Goddard management in using the new methodology. When we could not get information about process application, we followed standard SEL process-model activities.

We also noted other management and project-team concerns. Requirements and specifications change frequently during the development cycle in the FDD environment. This instability was of particular concern because the Cleanroom method is built on the precept of developing software right the first time. Another concern was that, given the difficulties encountered in the University of Maryland experiment about applying formal methods, how successfully could a classical Cleanroom approach be

applied? Finally, there was concern about the psychological effects of separating development and testing, specifically the inability of the developers to execute their code. We targeted all these concerns for our postproject analysis.

Project 1 lasted from January 1988 through September 1990. We separated the five team members into a three-person development team and a two-person test team. The development

team broke the total effort into six incremental builds of approximately 6,500 LOC each. An experimenter team consisting of NASA Goddard managers, SEL representatives, a technology advocate familiar with the IBM model, and the project leader monitored the overall process.

We modified the process in real time, as needed. For example, when we merged Cleanroom products into the standard FDD formal review and documentation activities, we had to modify both. We altered the design process to combine the use of state machines and traditional structured design. We also collected data for the monitoring team at various points throughout the project, although we tried to do this with as little disturbance as possible to the project team.

Analyzing and packaging results. The final steps in the QIP involve analyzing and packaging the process results. We found significant differences in effort distribution during development between the Cleanroom project and the baseline. Approximately six percent of the total project effort shifted from coding to design activities in the Cleanroom effort. Also, the baseline development teams traditionally spent approximately 85 percent of their coding effort writing code, 15 percent reading it. The Cleanroom team spent about 50 percent in each activity.

The primary goal of the first case study had been to improve reliability without increasing cost. Analysis showed a reduction in change rate of nearly 50 percent and a reduction in error rate of greater than a third. Although the expectation was for productivity equivalent to the baseline, the Cleanroom effort also improved in that area by approximately 50 percent. We also saw a decrease in rework, as defined by the amount of time spent correcting errors. Additional analysis of code reading revealed that three fourths of all errors uncovered were found by only one reader. This prompted a renewed emphasis on mul-

PROJECT RESULTS LED US TO EMPHASIZE PEER REVIEWS AND USE OF INDEPENDENT TESTING.

QUALITY IMPROVEMENT PARADIGM: FOUNDATION FOR IMPROVEMENT

The Quality Improvement Paradigm is an effective framework for conducting experiments and studies like those described in the main text. It is an experimental but evolutionary concept for learning and improvement.¹

The QIP has six steps:

1. Characterize the project and its environment.
2. Set quantifiable goals for successful project performance and improvement.
3. Choose the appropriate process models, supporting methods, and tools for the project.
4. Execute the processes, construct the products, collect and validate the prescribed data, and analyze the data to provide real-time feedback for corrective action.
5. Analyze the data to evaluate current practices, determine problems, record findings, and make recommendations for future process improvements.
6. Package the experience in the form of updated and refined models, and save the knowledge gained from this and earlier projects in an experience base for future projects.

The QIP uses two tools: the Goal/Question/Metric paradigm and the Experience Factory Organization.

GQM paradigm. The GQM paradigm is a mechanism used in the planning phase of the Quality Improvement Paradigm for defining and evaluating a set of operational goals using measurement.² It provides a systematic approach for tailoring and integrating goals with models of the software processes, products, and quality perspectives of interest, according to the specific needs of the project and organization.

You define goals in an operational, tractable way by refining them into a set of questions that extract appropriate information from the models. The questions, in turn, define the metrics needed to define and interpret the goals.

A goal-generation template helps in developing goals. The template specifies the essential elements: the object of interest (like product or process), the aspect of interest (like cost or ability

to detect defects), the purpose of the study (like assessment or prediction), the point of view from which the study is performed (like customer's or manager's), and the context in which the study is performed (like people-oriented or problem-oriented factors).

For example, two goals associated with the application of the Cleanroom method in the SEL were analysis of the Cleanroom process to characterize resource allocation from the project manager's point of view, and analysis of the Cleanroom product to characterize defects from the customer's point of view.

Experience Factory Organization. The Experience Factory Organization is an organizational structure that supports the activities specified in the QIP by continuously accumulating evaluated experiences, building a repository of integrated experience models that projects can access and modify to meet their needs.³ The Experience Factory extends project-development activities by providing systematic

learning and packaging of reusable experiences. It packages experiences by building informal, schematized, formal, and automated models and measures of software processes, products, and other forms of knowledge, and distributes them through consultation, documentation, and automated support.

While project organization follows an evolutionary process model that reuses packaged experiences, the Experience Factory provides the set of processes needed for learning, packaging, and storing the project organization's experience for reuse. The Experience Factory Organization represents the integration of these two functions.

REFERENCES

1. V. Basili, "Quantitative Evaluation of Software Engineering Methodology," Tech. Report TR-1519, CS Dept., Univ. of Maryland, College Park, July 1985.
2. V. Basili and H. Rombach, "The TAME Project: Towards Improvement-Oriented Software Environments," *IEEE Trans. Software Eng.*, June 1988, pp. 758-773.
3. V. Basili, "Software Development: A Paradigm for the Future," *Proc. Computer, IEEE CS Press, Los Alamitos, Calif.*, 1989.

multiple readers throughout the SEL environment.

We also examined the earlier concerns expressed by managers and the project team. The results showed increased effort in early requirements-analysis and design activities and a clearer set of in-line comments. This led to a better understanding of the whole system and enabled the project team to understand and accommodate changes with greater ease than was typical for that environment.

We reviewed the application of classical Cleanroom and noted successes and difficulties. The structure of independent teams and the emphasis on peer review during development was easy to apply. However, the devel-

opment team did have difficulty using the associated formal methods. Also, unlike the scheme in the classical Cleanroom method, the test team followed an approach that combined statistical testing with traditional functional testing.

Finally, the psychological effects of independent testing appeared to be negligible. All team members indicated high job satisfaction as well as a willingness to apply the method in future projects.

We packaged these early results in various reports and presentations, including some at the SEL's 1990 Software Engineering Workshop. As a reference for future SEL Cleanroom projects, we also began efforts to pro-

duce a document describing the SEL Cleanroom process model, including details on specific activities.⁶ (The completed document is now available to current Cleanroom projects.)

Second case study. The first case study showed us that we needed better training in the use of formal methods and more guidance in applying the testing approach. We also realized that experiences from the initial project team had to be disseminated and used.

Again, we followed the Quality Improvement Paradigm. We selected two projects: one similar to the initial Cleanroom project, Project 2A, and one more representative of the typical FDD contractor-support environment,

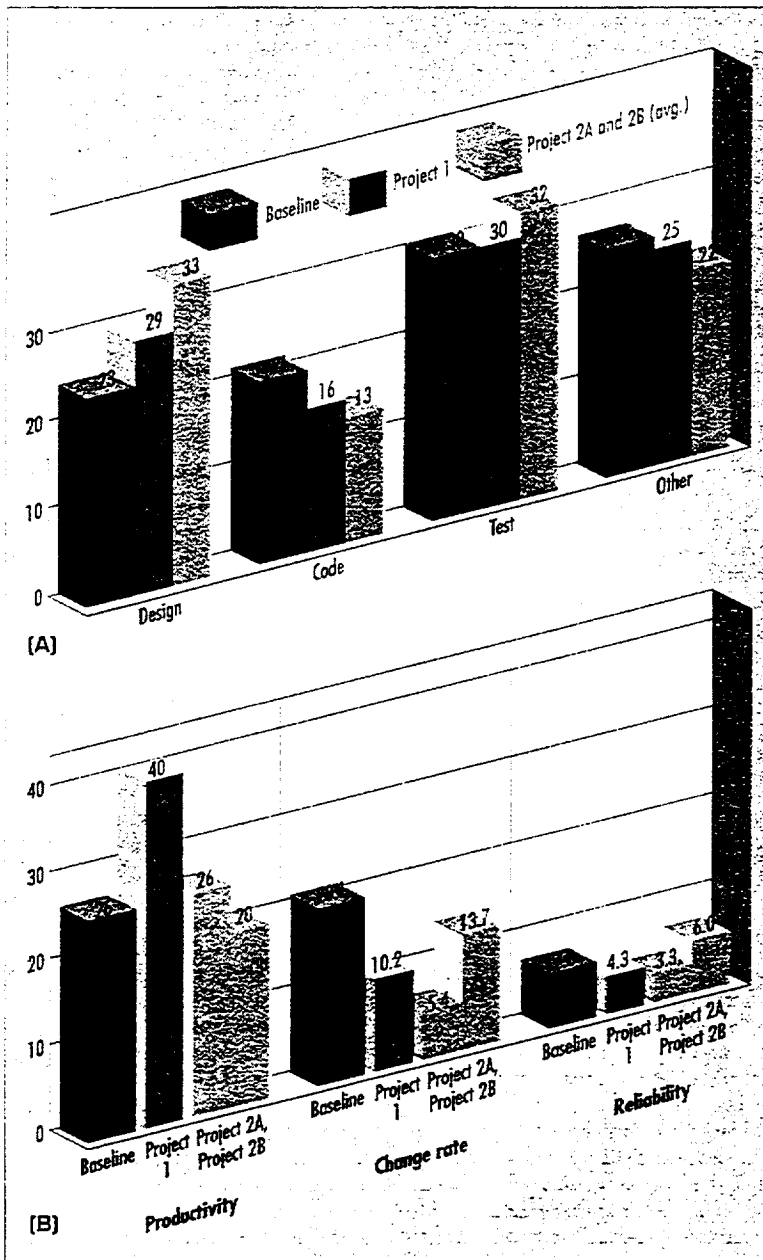


Figure 3. Measurement comparisons for two case studies investigating Cleanroom. The first case study involved one project, Project 1. The second case study involved two projects, Projects 2A and 2B. (A) Percentage of total development effort for various development activities, and (B) productivity in lines of code per day, change rate in changes per thousand lines of code, and reliability in errors per thousand lines of code.

Project 2B.

Project 2A involved a different subsystem of another attitude ground-support system. This subsystem focused on the processing of telemetry data, comprising 22,000 LOC. The project was staffed with four developers and two testers. Project 2B involved an entire mission attitude ground-support system, consisting of approximately 160,000 LOC. At its peak, it was staffed with 14 developers and four testers.

Setting goals and choosing processes. The second case study had two goals. One was to verify measures from the first study by applying the Cleanroom method to Project 2A, a project of similar size and scope. The second was to verify the applicability of Cleanroom on Project 2B, a substantially larger project but one more representative of the typical environment. We also wanted to further tailor the process model to the environment by using results from the first case study and applying more formal techniques.

Packages from the SEL Experience Factory (described in the box on p. 63) were available to support project development. These included an evolved training program, a more knowledgeable experimenter team to monitor the projects, and several in-process interactive sessions with the project teams. Although we had begun producing a handbook detailing the SEL Cleanroom process model, it was not ready in time to give to the teams at the start of these projects.

The project leader for the initial Cleanroom project participated as a member of the experimenter team, served as the process modeler for the handbook, and acted as a consultant to the current projects.

We modified the process according to the experiences of the Cleanroom team in the first study. Project 1's team had had difficulty using state machines in system design, so we changed the emphasis to Mills' box-structure algorithm.⁷ We also added a more extensive

TABLE 1
PROJECT COMPARISONS FOR SEL TECHNOLOGY EVALUATION

Evaluation aspect	Controlled experiments		Cleanroom case studies		
	Reading vs. testing	Cleanroom	Project 1	Project 2A	Project 2B
Team size	32 participants	Three-person development teams (10 experiment teams; five control teams); common independent tester	Three-person development team; two-person test team	Four-person development team; two-person test team	Fourteen-person development team; four-person test team
Project size and application	Small (145-365 LOC) sample Fortran programs	1,500 LOC, Fortran, electronic message system for graduate laboratory course	40,000 LOC, Fortran, flight-dynamics ground-support system	22,000 LOC, Fortran, flight-dynamics ground-support system	160,000 LOC, Fortran, flight-dynamics ground-support system
Results	Reading techniques appear more effective than testing techniques for fault detection	Cleanroom teams use fewer computer resources, satisfy requirements more successfully, and make higher percentage of scheduled deliveries	Project spends higher percentage of effort in design, uses fewer computer resources, and achieves better productivity and reliability than environment baseline	Project continues trend in better reliability while maintaining baseline productivity	Project reliability only slightly better than baseline while productivity falls below baseline

training program focusing on Cleanroom techniques, experiences from the initial Cleanroom team, and the relationship between the Cleanroom studies and the SEL's general goals. The instruction team included representatives from the SEL, members of the initial team, and Mills. Mills gave talks on various aspects of the methodology, as well as motivational remarks on the potential benefits of the Cleanroom method in the software community.

Project 2A ran from March 1990 through January 1992. Project 2B ran from February 1990 through December 1992. Again, we examined reliability, productivity, and process characteristics, comparing them to Project 1 results and the SEL baseline.

Analyzing and packaging results. As Figure 3 shows, there were significant differences between the two projects. Error and change rates for Project 2A continued to be favorable. Productivity rate, however, returned to the SEL baseline value. Error and change rates for Project 2B increased from Project 1 values, although they remained lower than SEL baseline numbers. Productivity, however, dropped below the baseline.

When we examined the effort distribution among the baseline and Projects 1, 2A, and 2B, we found a

continuing upward trend in the percentage of design effort, and a corresponding decrease in coding effort. Additional analysis indicated that although the overall error rates were below the baseline, the percentage of system components found to contain errors during testing was still representative of baseline projects developed in this environment. This suggests that the breadth of error distribution did not change with the Cleanroom method.

In addition to evaluating objective data for these two projects, we gathered subjective input through written and verbal feedback from project participants. In general, input from Project 2A team members, the smaller of the two projects, was very favorable, while Project 2B members, the larger contractor team, had significant reservations about the method's application. Interestingly, though, specific shortcomings were remarkably similar for both teams. Four areas were generally cited in the comments. Participants were dissatisfied with the use of design abstractions and box structures, did not fully accept the rationale for having no developer compilation, had problems coordinating information between developers and testers, and cited the need for a reference to the SEL Clean-

room process model.

Again, we packaged these results into various reports and presentations, which formed the basis for additional process tailoring.

Third case study. We have recently begun a third case study to examine difficulties in scaling up the Cleanroom method in the typical contractor-support environment and to verify previous trends and analyze additional tailoring of the SEL process model. We expect the study to complete in September.

In keeping with this goal, we again selected a project representative of the FDD contractor-support environment, but one that was estimated at 110,000 LOC, somewhat smaller than Project 2B. The project involves development of another entire mission attitude ground-support system. Several team members have prior experience with the Cleanroom method through previous SEL studies.

Experience Factory packages available to this project include training in the Cleanroom method, an experienced experimenter team, and the *SEL Cleanroom Process Model* (the completed handbook). In addition to modifying the process model according to the results from the first two case studies, we are

providing regularly scheduled sessions in which the team members and experimenters can interact. These sessions give team members the opportunity to communicate problems they are having in applying the method, ask for clarification, and get feedback on their activities. This activity is aimed at closing a communication gap that the contractor team felt existed in Project 2B.

The concepts associated with the QIP and its use of measurement have given us an evolutionary framework for understanding, assessing, and packaging the SEL's experiences.

Table 1 shows how the evolution of our Cleanroom study progressed as we used measurements from each experiment and case study to define the next experiment or study. The SEL Cleanroom process model has evolved on the basis of results packaged through earlier evaluations. Some aspects of the target methodology continue to evolve: Experimentation with formal methods has transitioned from functional decomposition and state machines to box-structure design and again to box-structure application as a way to abstract requirements. Testing has shifted from a combined statistical/functional approach, to a purely statistical approach based on operational scenarios. Our current case study is examining the effect of allowing developer compilation.

Along the way, we have eliminated some aspects of the candidate process; we have not examined reliability models, for example, since the environment does not currently have sufficient data to seed them. We have also emphasized some aspects. For example, we are conducting studies that focus on the effect of peer reviews and independent test teams for non-Cleanroom projects. We are also studying how to improve reading by developing reading techniques through off-line experimentation.

The SEL baseline used for comparison is undergoing continual evolution. Promising techniques are filtered into the development organization as general

process improvements, and corresponding measures of the modified process (effort distribution, reliability, cost) indicate the effect on the baseline.

The SEL Cleanroom process model has evolved to a point where it appears applicable to smaller projects (fewer than 50,000 LOC), but additional understanding and tailoring is still required for larger scale efforts. The model will continue to evolve as we gain more data from development projects. Measurement will provide baselines for comparison, identify areas of concern and improvement, and provide insight into the effects of

process modifications. In this way, we can set quantitative expectations and evaluate the degree to which goals have been achieved.

By adhering to the Quality Improvement Paradigm, we can refine the process model from study to study, assessing strengths and weaknesses, experiences, and goals. However, our investigation into the Cleanroom method illustrates that the evolutionary infusion of technology is not trivial and that process improvement depends on a structured approach of understanding, assessment, and packaging. ♦

ACKNOWLEDGMENTS

This work has been supported by NASA/GSFC contract NSG-5123. We thank all the members of the SEL team who have been part of the Cleanroom experimenter teams, the Cleanroom training teams, and the various Cleanroom project teams. We especially thank Frank McGarry, Rose Pajerski, Sally Godfrey, Ara Kouhadjian, Sharon Waligora, Harlan Mills, Michael Dyer, and Terry Baker for their efforts.

REFERENCES

1. V. Basili and R. Selby, "Comparing the Effectiveness of Software Testing Strategies," *IEEE Trans. Software Eng.*, Dec. 1987, pp. 1278-1296.
2. R. Linger, H. Mills, and B. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley, Reading, Mass., 1979.
3. H. Mills, M. Dyer, and R. Linger, "Cleanroom Software Engineering," *IEEE Software*, Sept. 1987, pp. 10-24.
4. R. Selby, Jr., V. Basili, and T. Baker, "Cleanroom Software Development: An Empirical Evaluation," *IEEE Trans. Software Eng.*, Sept. 1987, pp. 1027-1037.
5. L. Landis et al., "Recommended Approach to Software Development: Revision 3," Tech. Report SEL-81-305, Software Engineering Laboratory, Greenbelt, Md., 1992.
6. S. Green, *Software Engineering Laboratory (SEL) Cleanroom Process Model*, Tech. Report SEL-91-004, Software Engineering Laboratory, Greenbelt, Md., 1991.
7. H. Mills, "Stepwise Refinement and Verification in Box-Structured Systems," *IEEE Software*, June 1988, pp. 23-36.



Victor Basili is a professor of computer science at the Institute for Advanced Computer Studies at the University of Maryland at College Park. One of the founders and principals of the Software Engineering Laboratory, his interests include quantitative approaches for software

management, engineering, and quality assurance. He is on the editorial board of *Journal of Systems and Software*.

Basili received a BS in mathematics from Fordham College, an MS in mathematics from Syracuse University, and a PhD in computer science from the University of Texas at Austin. He is an IEEE fellow and a member of the IEEE Computer Society.

Scott Green is a senior software engineer in NASA Goddard's Flight Dynamics Division, where he is involved in the project management of ground-support systems and in leading software-engineering studies at the Software Engineering Laboratory.

Green received a BS in computer science from Loyola College.

Address questions about this article to Basili at CS Dept., University of Maryland, College Park, MD 20742; basili@cs.umd.edu; or to Green at NASA/GSFC, Code 552.1, Greenbelt, MD 20771; segreen@gsfcmail.nasa.gov.

SECTION 4—ADA TECHNOLOGY

The technical paper included in this section was originally prepared as indicated below.

- “Genericity Versus Inheritance Reconsidered: Self-Reference Using Generics,”
E. Seidewitz, *Proceedings of the Conference on Object-Oriented Programming
Systems, Languages, and Applications*, October 1994

**GENERICITY VERSUS INHERITANCE RECONSIDERED:
SELF-REFERENCE USING GENERICS
OOPSLA '94**

Ed Seidewitz
NASA Goddard Space Flight Center
Code 552.3
Greenbelt MD 20771
(301)286-7631
eseidewitz@gsfcmail.gsfc.nasa.gov

Abstract

As shown by the work of Bertrand Meyer, it is possible to simulate genericity using inheritance, but not vice-versa. This is because genericity is a parameterization mechanism with no way to deal with the polymorphic typing introduced using inheritance. Nevertheless, if we focus on the use of inheritance as an implementation technique, its key feature is the dynamic binding of self-referential operation calls. This turns out to be basically a parameterization mechanism that can in fact be simulated using generics and static binding. And for some applications this approach may actually be of more than academic interest.

Introduction

In his classic paper on "Genericity versus Inheritance", Bertrand Meyer concludes that inheritance cannot be simulated using genericity because genericity provides no mechanism for achieving the polymorphism of inheritance [Meyer 86]. This is, of course, true, since genericity is a parameterization mechanism, not a typing mechanism. However, as an *implementation* technique, rather than as a typing mechanism, the polymorphism of inheritance is primarily used to achieve the dynamic binding of self-referential calls to object operations (e.g., messages to `self` in Smalltalk).

This is not a minor point. Wegner and Zdonik state that "In a world without self-reference, inheritance reduces to invocation and inheritance hierarchies are simply tree-structured resource sharing hi-

erarchies. However, recursive definitions are just as fundamental for objects as for functions and procedures." [Wegner 88]. In effect, inheritance is not inheritance without self-reference. In this paper I will show that this crucial self-reference property of inheritance can, in fact, be simulated using genericity.

Cook and Palsberg define a denotational semantics of self-referential inheritance equivalent to the traditional operational semantics using dynamic binding [Cook 89]. They use a "wrapper" function to parameterize the super- and self-references of a class. These parameters are then "statically bound" using a fixed-point operation. Thus, self-reference becomes basically a parameterization problem, which can be handled quite well by generics.

The following three sections show in detail how this is done. The first section reviews the general issues of self-reference in the traditional inheritance mechanism. The next section shows how generics can be used to parameterize this self-reference. Finally, the third section extends this approach to also parameterize superclass reference.

The examples in this paper are written in Ada 9X, the proposed revision to the Ada language [Ada9X 94a] (likely to be approved in 1994). Ada 9X has powerful features for both genericity and object-oriented inheritance and is therefore an excellent real-world vehicle for the discussion here. I will introduce and describe the Ada 9X mechanisms for inheritance and genericity as necessary in the following. This should be sufficient for a self-contained reading of this paper, but it is by no means a complete overview of Ada 9X, or even its object-oriented features. For fuller discussions of Ada 9X, I refer the reader to the references [Ada9X 94a], [Ada9X 94b] and [Taft 93].

Inheritance

Hauk uses an instructive example to discuss the issues involved in inheritance and self-reference [Hauk 93]. This example is based on a class of objects that service hardware ports. One can output characters and lines to such ports, with the output of lines defined in terms of the output of characters. We define this class in Ada 9X using the following *package specification*:

```
package Port is

  type Object is tagged private;

  procedure Put(O: in out Object; C: in Character);
  procedure Put_Line(O: in out Object; L: in String);

private

  type Object is tagged record _ and record;

end Port;
```

In Ada 9X, encapsulation is achieved by defining abstract data types called *private types*. The type `Port.Object` is defined as a private type in the *visible part* of the package specification above, with its full definition given in the *private part*. Public *primitive operations* on this private type are also declared in the visible part of the package specification. The implementations of these operations are given in the corresponding *package body*, which we will get to in a moment.

The use of the keyword `tagged` in the definition of `Port.Object` signals the availability of the object-oriented features of *type extension* and *dispatching* for this type. For example, suppose we wish to define a subclass of ports that buffer their output. We can define this as an extension of `Port.Object`:

```
with Port;
package Buffered_Port is

  type Object(Size: Positive) is
    new Port.Object with private;

  procedure Flush(O: in out Object);

private

  type Object(Size: Positive) is new Port.Object with
    record
      Last: Natural := 0;
      Buffer: String(1..Size);
    end record;

end Buffered_Port;
```

The type `Buffered_Port.Object` is a *derived type* of `Port.Object` extended with the components required to implement a buffer. The *discriminant size* is used to set the maximum number of characters stored in the buffer. A derived type inherits the primitive operations of its *parent type*. In this case, `Buffered_Port.Object` inherits the operations `Put` and `Put_Line` from `Port.Object`. An additional operation, `Flush`, is defined solely on the type `Buffered_Port.Object`.

Derived types are distinct types from their parent types. Thus, given the declarations:

```
P: Port.Object;
B: Buffered_Port.Object;
```

the following assignment is *illegal*:

```
P := B;    -- Type mismatch!
```

even though `Buffered_Port.Object` is derived from `Port.Object`. The following explicit conversion is legal:

```
-- An object of type Buffered_Port.Object can be
-- converted to type Port.Object
P := Port.Object(B)
```

but the converted value is of type `Port.Object`, and the extension components in `B` are lost.

Ada 9X separates polymorphism from the basic tagged type construct through the concept of *class-wide types*. For example, there is a class-wide type denoted `Port.Object'Class` rooted in the tagged type `Port.Object`. A class-wide type includes all values of all types in the *derivation class* of its root tagged type. The derivation class of a tagged type includes the type and all *descendant* types derived from it.

Due to the availability of type extension, the size of a value of a class-wide type cannot generally be determined at compile time. Therefore, polymorphic variables in Ada 9X generally contain pointers to class-wide types. Pointer types in Ada are known as *access types*. Thus, given the following declarations:

```
type Port_Pointer is access Port.Object'Class;
type Buffered_Port_Pointer is access
  Buffered_Port.Object'Class;
```

```
PP: Port_Pointer;
BP: Buffered_Port_Pointer
```

the following assignment is *legal*:

```
-- Pointer to Port.Object'Class can point to
-- Buffered_Port'Class object
PP := BP;
```

because the derivation class of `Buffered_Port.Object` is contained in the derivation class of `Port.Object`.

In addition to allowing polymorphic variables, class-wide types also provide the mechanism for polymorphic dynamic binding of operations. Each value of a tagged type has a *tag* that identifies the dynamic type of that value. When a primitive operation of a tagged type is passed a value of the corresponding class-wide type (which may actually be a value of any type derived from the root tagged type), the operation *dispatches* to the implementation identified by the tag of the value. For example, given the above declarations:

```
-- Note that ".all" dereferences pointers
Port.Put(PP.all,C);      -- Bound to Put implementation
                        -- in body of Port

PP := BP;
Port.Put(PP.all,C);      -- Bound to Put implementation
                        -- in body of Buffered_Port
```

The second call to `Port.Put` is dynamically bound to `Buffered_Port.Put`, because the tag of the object pointed to by `PP` after the assignment indicates type `Buffered_Port.Object`.

Now let's turn to the body of package `Port`. This body contains the implementations of the two operations on type `Port.Object`:

```
package body Port is

  procedure Put(O: in out Object; C: in Character) is --
  end Put;

  procedure Put_Line(O: in out Object; L: in String) is
  begin
    for I in L'Range loop
      Put(Object'Class(O),L(I)); -- Redispatching call
    end loop;
  end Put_Line;

end Port;
```

Note the use of the conversion `Object'Class(O)` in the call above to the procedure `Put`. This conversion causes the call to `Put` to be dynamically bound, depending on the dynamic tag of the argument `O`. This is known as a *redispatching* call in Ada 9X, and it has the same effect as the use of *self* in Smalltalk

[Goldberg 83] or *this* in C++ (for a virtual function) [Stroustrup 91].

The use of redispatching in the implementation of `Put_Line` makes the implementation of type `Port.Object` *self-referential*. This self-reference is very important for the implementation of the operations of the derived type `Buffered_Port.Object`. The body of package `Buffered_Port` must, of course, include the implementation of the new operation `Flush`. In addition, the implementation of procedure `Put` inherited from `Port.Object` must be *overridden* with a new implementation that handles the buffering required for a `Buffered_Port.Object`:

```
package body Buffered_Port is

  procedure Put(O: in out Object; C: in Character) is
  begin
    O.Last := O.Last + 1;
    O.Buffer(O.Last) := C;
    if O.Last = O.Size then
      Flush(Object'Class(O)); -- Redispatching call
    end if;
  end Put;

  procedure Flush(O: in out Object) is
  begin
    for I in 1..O.Last loop
      Port.Put(Port.Object(O),O.Buffer(I));
    end loop;
    O.Last := 0;
  end Flush;

end Buffered_Port;
```

Note the conversion `Port.Object(O)` in the statically bound call to the parent operation `Port.Put` in the implementation of `Flush`.

Since the procedure `Put_Line` is not overridden in the body of package `Buffered_Port`, its implementation is inherited without change. This is shown diagrammatically in Figure 1, where the shading indicates that there is no implementation for `Put_Line`

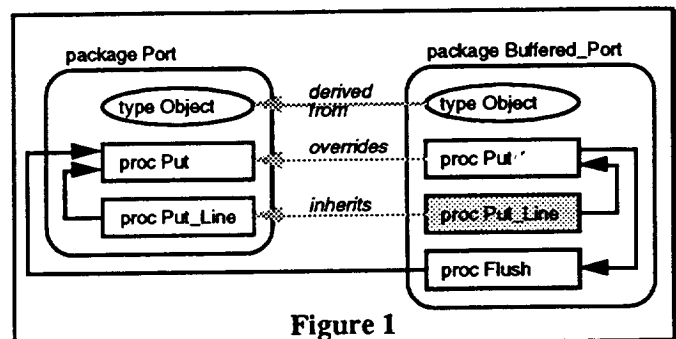


Figure 1

physically included in package `Buffered_Port`. Figure 1 also shows that the actual implementation for `Put_Line` in package `Port` makes a call on `Port.Put`. However, when this implementation is inherited in package `Buffered_Port`, the redispaching call to `Port.Put`, when passed a value of type `Buffered_Port.Object`, will now be dynamically bound to the overriding implementation of `Buffered_Port.Put`. Thus, the characters in a line are all properly buffered, even though the implementation of procedure `Put_Line` has not changed.

Genericity

Consider again the hardware port example from the last section. We wish to implement the same `Port.Object` private type, with the same visible operations, but without the use of redispaching. Nevertheless, we wish to retain the ability to redirect the binding of self-referential calls in operations inherited by a descendant of `Port.Object`. To do this we make this binding explicit using a generic package nested within the specification of package `Port`:

```
package Port is
    type Object is tagged private;
    procedure Put(O: in out Self; C: in Character);
    procedure Put_Line(O: in out Self; L: in String);

    generic
        type Self(<>) is new Object with private;
    package Operations is
        procedure Do_Put(O: in out Self; C: in Character);
        procedure Do_Put_Line(O: in out Self; L: in String);
    end Operations;

private
    type Object is tagged record ... end record;

end Port;
```

While the type `Port.Object` retains its operations `Put` and `Put_Line`, the actual implementation of these operations are moved to the inner generic package `Port.Operations`. This generic package is parameterized by the type `Self`, which must be a descendant of `Port.Object` (or `Port.Object` itself). As a descendant of `Port.Object`, any actual type bound to the parameter `Self` will have `Put` and `Put_Line` operations. This binding of the parameter `Self` will be used in the implementation of the operations `Do_Put` and `Do_Put_Line` to replace any self-

referential redispaching calls.

The generic package `operations` is nested inside `Port` so that its body has visibility to the full definition of the private type `Port.Object`. This allows the subprogram `Do_Put` to be implemented the same way as `Port.Put` would have been in the last section (if we had actually shown it!). The implementation of `Do_Put_Line` is also similar to the implementation of `Port.Put_Line` in the last section, but with a crucial difference:

```
package body Port is
    package body Operations is

        procedure Do_Put
            (O: in out Self; C: in Character) is
        ... end Put;

        procedure Do_Put_Line(O: in out Self; L: in String) is
        begin
            for I in L'Range loop
                Put(O, L(I));    -- Statically-bound call
            end loop;
        end Do_Put_Line;

    end Operations;

    package Self_Operations is
        new Operations(Port.Object);

    procedure Put(O: in out Self; C: in Character)
        renames Self_Operations.Do_Put;

    procedure Put_Line(O: in out Self; L: in String)
        renames Self_Operations.Do_Put_Line;

end Port;
```

In place of the redispaching call in the implementation of `Put_Line` there is now a *statically-bound* call in procedure `Do_Put_Line` to the operation `Put` on type `Self`. Rather than using redispaching, self-reference is achieved by instantiating the generic package `operations` in the body of package `Port`. This instantiation effectively provides the fixed-point operation of Cook and Palsberg.

The package `Self_Operations` is an instantiation of the generic package `operations` with type `Port.Object` used for the parameter `Self`. The procedures `Put` and `Put_Line` are then simply renamings of the real implementations from `Self_Operations` (which have the correct argument type profiles, since `Self` is `Port.Object` for `Self_Operations`!). The implementation of `Self_Operations.Do_Put_Line` contains a call to the operation

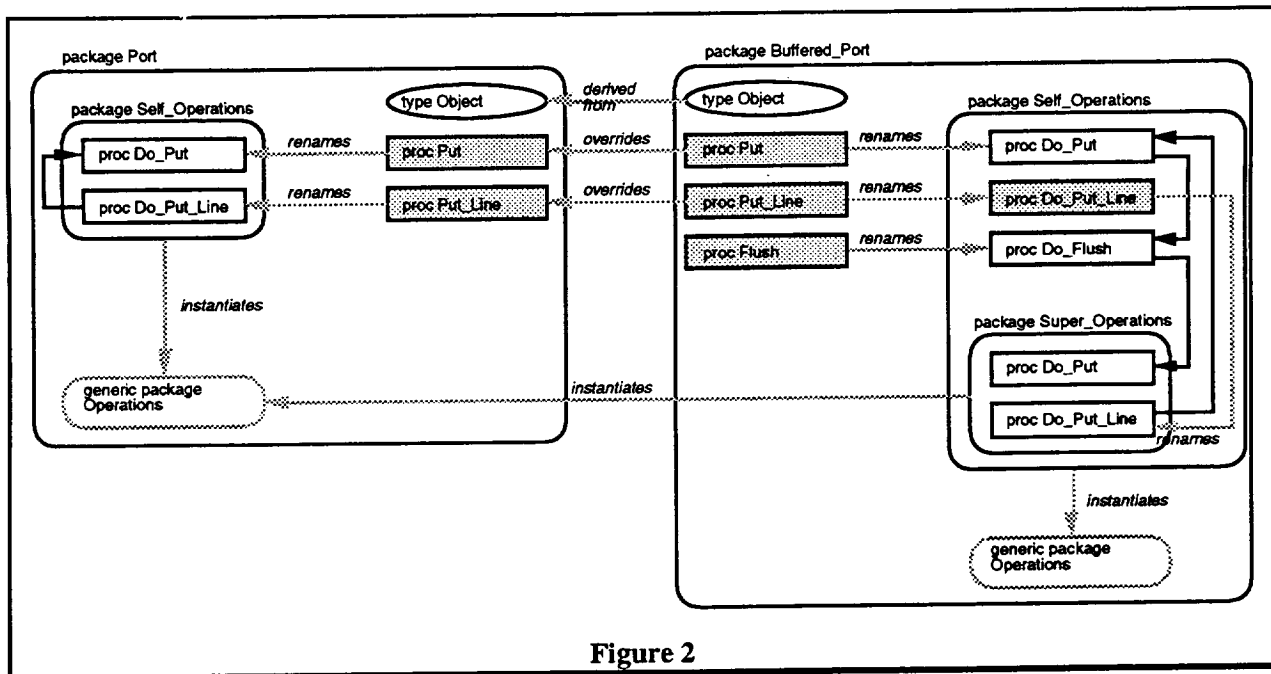


Figure 2

Put for the generic type parameter *self*. Since the type parameter *self* is bound to *Port.Object* for this instantiation, its *Put* operation is simply *Port.Put*, which is a renaming of *Self_Operations.Do_Put*. Thus *Port.Put_Line* self-referentially calls *Self_Operations.Do_Put*, as shown in Figure 2.

Note, however, that *Port.Put_Line* now makes a *statically-bound* call to *Port.Put*. Thus this call will not be automatically redirected to *Buffered_Port.Put* in the inherited operation *Buffered_Port.Put_Line*. Instead, we must instantiate the generic package *Port.Operations* *differently* for the implementation of the *Buffered_Port* operations, so as to achieve the correct bindings. To see how this is done, let's turn next to the implementation of *Buffered_Port.Object* using the our new approach.

As we did with package *Port*, we include a nested generic package within package *Buffered_Port*:

```
with Port;
package Buffered_Port is

  type Object(Size: Positive) is
    new Port.Object with private;
  procedure Flush(O: in out Self);

  generic
    type Self(<>) is new Object with private;
```

```
package Operations is
  procedure Do_Put(O: in out Self; C: in Character);
  procedure Do_Put_Line(O: in out Self; L: in String);
  procedure Do_Flush(O: in out Self);
end Operations;
```

private

```
type Object(Size: Positive) is new Port.Object with
  record
    Last: Natural := 0;
    Buffer: String(1..Size);
  end record;
```

end Buffered_Port;

Note that inner generic package *Buffered_Port.Operations* contains implementations for the inherited operations *Put* and *Put_Line* as well as the new buffered port operation *Flush*:

package body Buffered_Port is

package body Operations is

```
package Super_Operations is
  new Port.Operations(Self);
```

```
procedure Do_Put(O: in out Self; C: in Character) is
begin
  O.Last := O.Last + 1;
  O.Buffer(O.Last) := C;
  if O.Last = O.Size then
    Flush(O);      -- Statically-bound call
  end if;
end Put;
```

```
procedure Do_Put_Line(O: in out Self; L: in String)
renames Super_Operations.Do_Put_Line;
```

```

procedure Do_Flush(O: in out Object) is
begin
  for I in 1..O.Last loop
    Super_Operations.Do_Put(O,O.Buffer(I));
    -- Statically-bound call
  end loop;
  O.Last := 0;
  and Flush;

end Operations;

package Self_Operations is
  new Buffered_Port.Operations(Buffered_Port.Object);

procedure Put(O: in out Object; C: in Character)
  renames Self_Operations.Do_Put;

procedure Put_Line(O: in out Object; L: in String)
  renames Self_Operations.Do_Put_Line;

procedure Flush(O: in out Object)
  renames Self_Operations.Do_Flush;

end Buffered_Port;

```

Note the nested instantiation of `Port.Operations` within `Buffered_Port.Operations`, passing along the correct binding for `self`.

As shown in Figure 2, the instantiation `Buffered_Port.Self_Operations` appropriately redirects the self-referential calls to `Put` and `Flush` to the implementations as required. The nested instantiation of `Port.Operations` within `Buffered_Port.Self_Operations` assures that even references to `Put` in `Buffered_Port.Super_Operations.Do_Put_Line` now call `Buffered_Port.Self_Operations.Do_Put`.

Mixins

The wrapper functions of Cook and Palsberg parameterize *both* the super- and self-references of a class [Cook 89]. In the last section we used generics to parameterize the self-references. An extension of this approach can be used to parameterize superclass references as well.

To do this, we first turn the package defining the subclass type into a generic package with the superclass type as a generic parameter. Such a generic package provides an independent increment of functionality that can be added on to any appropriate superclass type. We will call such a package a *mixin*, since its functionality can be “mixed into” the superclass. The term “mixin” comes originally from the LISP-based Flavors system [Moon 86] and is usually used in conjunction with multiple inheritance. The

mixins we will define here are closer in spirit to the generalized concept proposed by Bracha and Cook [Bracha 90]. (See also the Ada 9X Rationale [Ada9X 94b] for a discussion of using generics as mixins in Ada 9X; I have also previously described how mixins can even be created in non-object-oriented Ada 83 [Seidewitz 92].)

For example, consider the buffered port class. We can turn this class into a mixin by replacing its superclass dependency on the port class with a generic parameter:

```

generic

  type Element is private;
  type Super(<>) is abstract tagged private;

package Buffer_Mixin is

  type Object(Size: Positive) is abstract new Super
  with private;

  generic
    type Self(<>) is new Object with private;
    with procedure Super_Put
      (O: in out Self; E: in Element);
    with procedure Self_Flush(O: in out Self);
  package Operations is
    procedure Do_Put(O: in out Self; E: in Element);
    procedure Do_Flush(O: in out Self);
  end Operations;

private

  type Element_Array is
    array (Positive range <>) of Element;

  type Object(Size: Positive) is
    abstract new Super with
    record
      Last: Natural := 0;
      Buffer: Element_Array(1..Size);
    end record;

end Buffered_Port;

```

The type parameter `super` provides the required parameterization of the superclass type. The type `Buffer_Mixin.Object` is then derived from this generic parameter. Since we needed to make this a generic package anyway, the buffer mixin is further generalized above by using the generic type parameter `Element` (which does not need to be tagged) in place of `Character`.

Note that the type parameter `super` is declared to be *abstract*. This means that the actual type used for this parameter may be an *abstract type* (though it may also be non-abstract). It is illegal to create ob-

jects of an abstract type, though there may be objects of non-abstract descendants of the abstract type. Further, an abstract type may have *abstract operations* that have no implementations (these are equivalent to pure virtual functions in C++ [Stroustrup 91] or deferred routines in Eiffel [Meyer 88]). Non-abstract descendants of an abstract type must override all abstract operations with non-abstract implementations. The type `Buffer_Mixin.Object` is also declared to be abstract, since it may inherit abstract operations from `Super`.

The type parameter `Super` is also not constrained to be a descendant of any known type. Therefore, within the body of the generic package, there are no primitive operations guaranteed to be available for `Super` (except for some basic operations like equality, but that's a technicality). Since there are no known operations to be inherited from `Super`, and no other operations are defined for it, the type `Buffer_Mixin.Object` also has no known primitive operations. Instead, this type only provides a basis for defining the implementations of the operations given in the generic package `Buffer_Mixin.Operations`.

As before, the generic package `Operations` is parameterized by the derived type parameter `Self`. Now, however, there are no known primitive operations to be inherited from `Buffer_Mixin.Object`. Instead, the only operations on `Self` are those that are explicitly given as generic parameters, in this case `Super_Put` and `Self_Flush`. As the names indicate, the `Super_Put` parameter is intended to provide the superclass `Put` operation, while the `Self_Flush` parameter provides the self-referential `Flush` operation. Thus, this generic clause defines the *complete* inheritance interface for the buffer mixin. (As will become clearer in a moment, the `Super_Put` operation is defined on the type `Self` rather than `Super` to ensure the correct binding of any self-referential calls it may make.)

Calls to the operations given by `Super_Put` and `Self_Flush` are now the *only* external calls that can be made on type `Self` in the implementations of `Do_Put` and `Do_Flush`:

```
package body Buffer_Mixin is
```

```
package body Operations is

  procedure Do_Put(O: in out Self; E: in Element) is
  begin
    O.Last := O.Last + 1;
    O.Buffer(O.Last) := E;
    if O.Last = O.Size then
      Self_Flush(O); -- Statically-bound call
    end if;
  end Put;

  procedure Do_Flush(O: in out Self) is
  begin
    for I in 1..O.Last loop
      Super_Put(O,O.Buffer(I));
    end loop;
    O.Last := 0;
  end Flush;

end Operations;

end Buffer_Mixin;
```

Note that this buffer mixin does not define a `Do_Put_Line` operation. This is because a mixin should represent a discrete increment of functionality, and the ability to put a line is not really part of the buffering functionality as defined here.

As defined in the previous sections, the port class does not have any superclass. However, for consistency, we can also turn this class into a mixin:

```
generic
  type Super(<>) is abstract tagged private;

package Port_Mixin is

  type Object is abstract new Super with private;

  generic
    type Self(<>) is new Object with private;
    with procedure Self_Put
      (O: in out Self; C: in Character);
  package Operations is
    procedure Do_Put(O: in out Self; C: in Character);
    procedure Do_Put_Line(O: in out Self; L: in String);
  end Operations;

private
  type Object is abstract new Super with
    record ... end record;

end Port;

package body Port_Mixin is

  package body Operations is

    procedure Do_Put(O: in out Self; C: in Character) is
    ... end Put;

    procedure Do_Put_Line(O: in out Self; L: in String) is
    begin
      for I in L'Range loop
```

```

        Self_Put(O,L(I)); -- Statically-bound call
    end loop;
    end Put_Line;

end Operations;

end Port_Mixin;

```

Even though the hardware port functionality does not require any superclass operations, this mixin allows such functionality to be freely mixed in as part of any class implementation.

Note that there is no typing relationship at all between the port and buffer mixins. Mixins provide incremental implementation completely independently of problem-domain typing relationships. As a complement to these mixins, we can define a set of abstract types that capture typing relationships completely independently of implementation details.

For example, we can use two abstract types to define the supertype/subtype relationship between ports and buffered ports:

```

package Port_Types is

    type Port is abstract tagged null record;
    procedure Put(O: in out Port; C: in Character) is
        abstract;
    procedure Put_Line(O: in out Port; L: in String) is
        abstract;

    type Buffered_Port is
        abstract new Port with null record;
    procedure Flush(O: in out Buffered_Port) is abstract;

end Port_Types;

```

For simplicity, this one package defines both abstract types, though they could equally well have been defined in separate packages.

To actually implement the port and buffered port classes, we need to bring together the functionality implemented in the port and buffer mixins with the type hierarchy defined by the port and buffered port abstract types. The following shows how this is done for the buffered port class:

```

with Port_Types, Port_Mixin, Buffer_Mixin;
package Buffered_Port is

    type Object is
        new Port_Types.Buffered_Port with private;

private

    package Port_Implementation is
        new Port_Mixin(Port_Types.Buffered_Port);

    package Buffered_Port_Implementation is
        new Buffer_Mixin
            (Character, Port_Implementation.Object);

    type Object is
        new Buffered_Port_Implementation.Object with
            null record;

end Buffered_Port;

```

The instantiations of the two mixins incrementally builds the implementation of the type `Buffered_Port.Object`.

As shown in Figure 3, the instantiation `Port_Implementation` adds port-related components to the type `Port_Types.Buffered_Port`

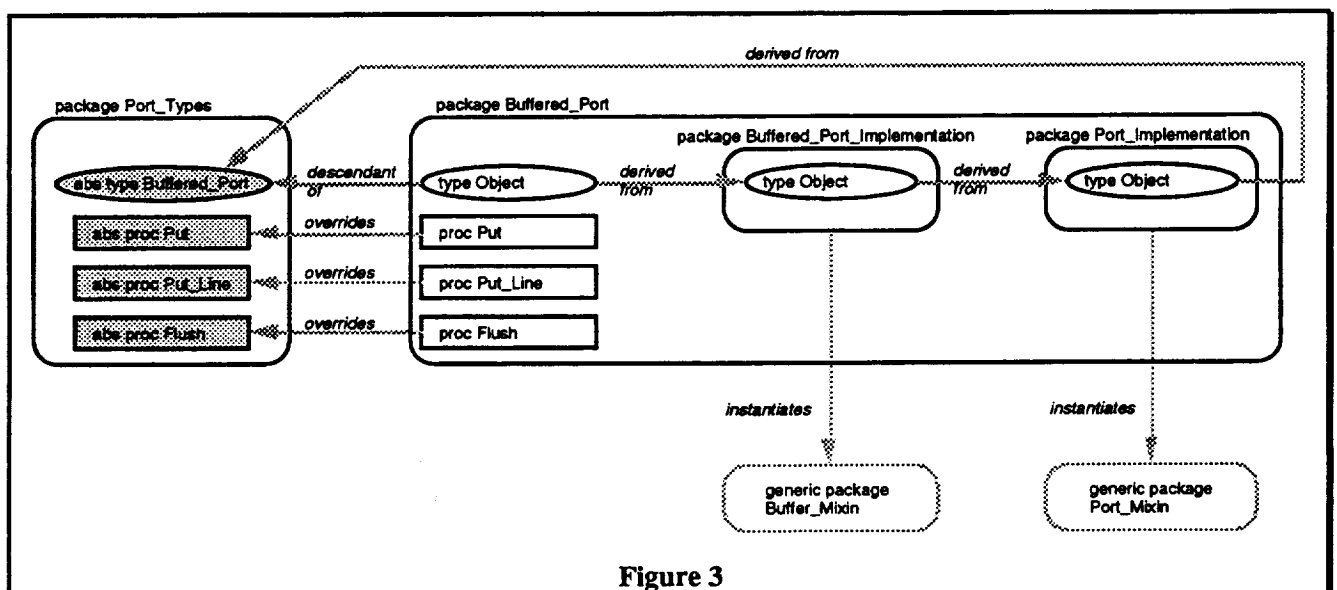


Figure 3

(which has no components itself), producing the type `Port_Implementation.Object` (this is also an example of why we need to allow mixin generics to be instantiated with abstract types). The instantiation `Buffered_Port_Implementation` then extends the type `Port_Implementation.Object` with buffer-related components, producing the type `Buffered_Port_Implementation.Object`. The full definition of `Buffered_Port.Object` is a *null extension* of `Buffered_Port_Implementation.Object`.

The *partial view* of `Buffered_Port.Object` given in the visible part of package `Buffered_Port` declares this type to be a descendant of `Port_Types.Port`. The full definition of `Buffered_Port.Object` given in the private part of the package is indeed a descendant of the abstract type `Port_Types.Port` via the type extensions resulting from the two mixin instantiations and the final null extension. As such, it inherits the three abstract operations `Put`, `Put_Line` and `Flush`. However, `Buffered_Port.Object` is *not* declared to be abstract and so must provide implementations for these inherited operations.

The implementations of the `Buffered_Port.Object` operations are, of course, given in the body of package `Buffered_Port`, using the operations generic packages from the port and buffer mixins:

```
package body Buffered_Port is

  package Port_Operations is
    new Port_Implementation.Operations
    (Buffered_Port.Object, Put);

  package Buffered_Port_Operations is
    new Buffered_Port_Implementation.Operations
    ( Bufferd_Port.Object,
      Port_Operations.Do_Put,
      Flush);

  procedure Put(O: in out Object; C: in Character)
    renames Buffered_Port_Operations.Do_Put;

  procedure Put_Line(O: in out Object; L: in String)
    renames Port_Operations.Do_Put_Line;

  procedure Flush(O: in out Object)
    renames Buffered_Port_Operations.Do_Flush;

end Buffered_Port;
```

Since the buffer implementation is now independent of the port implementation, *both* operations generic packages must be instantiated here. The in-

stantiation of `Port_Implementation.Operations` uses operation `Buffered_Port.Put` for the self-referential `Self_Put` generic parameter. The instantiation of `Buffered_Port_Implementation.Operations` uses operation `Buffered_Port.Flush` for the self-referential `Self_Flush` parameter. However, it uses the operation `Port_Operations.Do_Put`, *not* `Buffered_Port.Put`, for the superclass operation `Super_Put`. (This also shows why superclass operations must be parameters of the *inner* generic package operations in a mixin.)

The actual `Buffered_Port.Object` operations are once again defined as renamings of subprograms from the instantiated operations packages. Note, however, that `Put_Line` is taken from `Port_Operations`, *not* `Buffered_Port_Operations`, since the buffer mixin does not implement a `Put_Line` operation. Nevertheless, the generic instantiations insure that `Buffered_Port.Put_Line` is implemented with a proper self-referential call to `Buffered_Port_Operations.Do_Put` (the reader can trace how this happens using Figure 4).

Conclusion

The use of generics for the static-binding of self-referential calls is at least of academic interest in the comparison of inheritance and genericity. However, since the generic approach can be a bit cumbersome, one may ask if it has any practical application. In fact, there are some good reasons to consider this approach:

1. Experience has shown that the common use of self-reference with inheritance can make an object-oriented program difficult to understand and change (see, for example, [Taenzer 89], [Leijter 92], [Wild 92] and [Wild 93]). The generic approach gives the programmer much more precise control about when and where these self-referential bindings are made and thus makes the use and intent of self-reference more apparent to the maintainer.
2. In many safety-critical applications (such as avionics software), any "dynamic" construct (dynamic memory allocation, dynamics binding, etc.) is regarded with suspicion. This is because

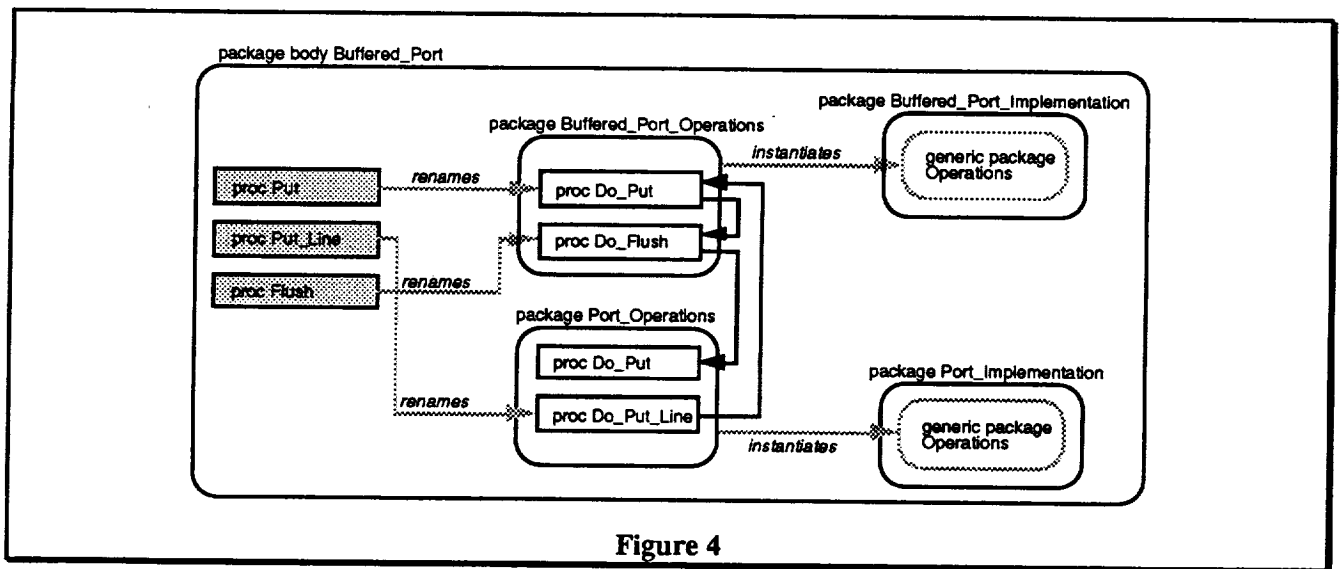


Figure 4

such features make it much harder to verify that a program meets stringent safety requirements. The generic approach provides self-reference and deferred operation implementation with fully static binding.

3. For a generic mixin, the generic clause of the inner `operations` generic package effectively gives a complete “typing” of the inheritance interface. That is, it explicitly lists all operations required from the superclass and all operations called self-referentially. As described by Hauk, such a complete typing allows the type-safe replacement of a superclass implementation during class library maintenance [Hauk 93] (see also [Gibbs 90] on the issues of modifying class hierarchies). For example, in the `Buffered_Port` implementation given in the last section, the use of the `Port_Mixin` could be easily replaced with a different implementation of the hardware port functionality, so long as it provided the `Put` operation needed by `Buffer_Mixin`. Such a replacement requires no

changes to the implementation of the buffering functionality, nor any changes to the clients of `Buffered_Port`. For that matter, it would be equally easy to replace the `Buffer_Mixin` with a different implementation of the buffering functionality.

Meyer was indeed correct in concluding that genericity cannot be used to fully simulate inheritance. However, inheritance is a much more expansive mechanism than genericity, and thus the comparison with genericity is not entirely fair. We can decompose the inheritance mechanism as type extension plus polymorphic typing plus self-reference. Genericity is only comparable to the parameterization-oriented effect of self-reference in the inheritance mechanism. As we have seen in this paper, inheritance actually *can* be simulated by type extension plus polymorphic typing *plus genericity*, and that the generic approach actually has some potential advantages.

References

- [Ada9X 94a] *Ada 9X Reference Manual (Draft Version 5.0)*, ANSI/ISO/IEC DIS 8652, Ada Mapping/Revision Team, Intermetrics, June 1994
- [Ada9X 94b] *Ada 9X Rationale (Draft Version 5.0)*, Ada Mapping/Revision Team, Intermetrics, June 1994
- [Bracha 90] G. Bracha and W. Cook, "Mixin-based Inheritance", *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications / European Conference on Object-Oriented Programming*, SIGPLAN Notices, October 1990
- [Cook 89] W. Cook and L. Palsberg, "A Denotational Semantics of Inheritance and its Correctness", *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications*, SIGPLAN Notices, October 1989
- [Gibbs 90] S. Gibbs, D. Tsichritzis, E. Casais, O. Nierstrasz and X. Pintado, "Class Management for Software Communities", *Communications of the ACM*, September 1990
- [Goldberg 93] A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, 1983
- [Hauck 93] F. Hauck, "Inheritance Modeled with Explicit Bindings: An Approach to Typed Inheritance", *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications*, SIGPLAN Notices, September/October 1993
- [Leijter 92] M. Leijter, S. Meyers and S. P. Reiss, "Support for Maintaining Object-Oriented Programs", *IEEE Transactions on Software Engineering*, December 1992
- [Meyer 86] B. Meyer, "Genericity versus Inheritance", *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications*, SIGPLAN Notices, November 1986
- [Meyer 88] B. Meyer, *Object-Oriented Software Construction*, Prentice-Hall, 1988
- [Moon 86] D. A. Moon, "Object-Oriented Programming with Flavors", *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications*, SIGPLAN Notices, November 1986
- [Seidewitz 92] E. Seidewitz, "Object-Oriented Programming with Mixins in Ada", *Ada Letters*, March/April 1992
- [Stroustrup 91] B. Stroustrup, *The C++ Programming Language (2nd ed.)*, Addison-Wesley, 1991
- [Taenzer 89] D. Taenzer, M. Ganti and S. Podar, "Object-Oriented Software Reuse: The Yoyo Problem", *Journal of Object-Oriented Programming*, September/October 1989
- [Taft 93] T. Taft, "Ada 9X: From Abstraction -Oriented to Object-Oriented", *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications*, SIGPLAN Notices, October 1993.
- [Wegner 88] P. Wegner and S. B. Zdonik, "Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like," *Proceedings of the European Conference on Object-Oriented Programming*, LNCS 322, Springer-Verlag, August 1988
- [Wilde 92] N. Wilde and R. Huitt, "Maintenance Support for Object-Oriented Programs", *IEEE Transactions on Software Engineering*, December 1992
- [Wilde 93] N. Wilde, P. Matthews and R. Huitt, "Maintaining Object-Oriented Software", *IEEE Software*, January 1993

STANDARD BIBLIOGRAPHY OF SEL LITERATURE

The technical papers, memorandums, and documents listed in this bibliography are organized into two groups. The first group is composed of documents issued by the Software Engineering Laboratory (SEL) during its research and development activities. The second group includes materials that were published elsewhere but pertain to SEL activities.

SEL-ORIGINATED DOCUMENTS

SEL-76-001, *Proceedings From the First Summer Software Engineering Workshop*, August 1976

SEL-77-002, *Proceedings From the Second Summer Software Engineering Workshop*, September 1977

SEL-78-005, *Proceedings From the Third Summer Software Engineering Workshop*, September 1978

SEL-78-006, *GSFC Software Engineering Research Requirements Analysis Study*, P. A. Scheffer and C. E. Velez, November 1978

SEL-78-007, *Applicability of the Rayleigh Curve to the SEL Environment*, T. E. Mapp, December 1978

SEL-78-302, *FORTTRAN Static Source Code Analyzer Program (SAP) User's Guide (Revision 3)*, W. J. Decker, W. A. Taylor, et al., July 1986

SEL-79-002, *The Software Engineering Laboratory: Relationship Equations*, K. Freburger and V. R. Basili, May 1979

SEL-79-004, *Evaluation of the Caine, Farber, and Gordon Program Design Language (PDL) in the Goddard Space Flight Center (GSFC) Code 580 Software Design Environment*, C. E. Goorevich, A. L. Green, and W. J. Decker, September 1979

SEL-79-005, *Proceedings From the Fourth Summer Software Engineering Workshop*, November 1979

SEL-80-002, *Multi-Level Expression Design Language-Requirement Level (MEDL-R) System Evaluation*, W. J. Decker and C. E. Goorevich, May 1980

SEL-80-005, *A Study of the Musa Reliability Model*, A. M. Miller, November 1980

SEL-80-006, *Proceedings From the Fifth Annual Software Engineering Workshop*, November 1980

SEL-80-007, *An Appraisal of Selected Cost/Resource Estimation Models for Software Systems*, J. F. Cook and F. E. McGarry, December 1980

SEL-80-008, *Tutorial on Models and Metrics for Software Management and Engineering*, V. R. Basili, 1980

SEL-81-011, *Evaluating Software Development by Analysis of Change Data*, D. M. Weiss, November 1981

SEL-81-012, *The Rayleigh Curve as a Model for Effort Distribution Over the Life of Medium Scale Software Systems*, G. O. Picasso, December 1981

SEL-81-013, *Proceedings of the Sixth Annual Software Engineering Workshop*, December 1981

SEL-81-014, *Automated Collection of Software Engineering Data in the Software Engineering Laboratory (SEL)*, A. L. Green, W. J. Decker, and F. E. McGarry, September 1981

SEL-81-101, *Guide to Data Collection*, V. E. Church, D. N. Card, F. E. McGarry, et al., August 1982

SEL-81-104, *The Software Engineering Laboratory*, D. N. Card, F. E. McGarry, G. Page, et al., February 1982

SEL-81-110, *Evaluation of an Independent Verification and Validation (IV&V) Methodology for Flight Dynamics*, G. Page, F. E. McGarry, and D. N. Card, June 1985

SEL-81-305, *Recommended Approach to Software Development*, L. Landis, S. Waligora, F. E. McGarry, et al., June 1992

SEL-81-305SP1, *Ada Developers' Supplement to the Recommended Approach*, R. Kester and L. Landis, November 1993

SEL-82-001, *Evaluation of Management Measures of Software Development*, G. Page, D. N. Card, and F. E. McGarry, September 1982, vols. 1 and 2

SEL-82-004, *Collected Software Engineering Papers: Volume 1*, July 1982

SEL-82-007, *Proceedings of the Seventh Annual Software Engineering Workshop*, December 1982

SEL-82-008, *Evaluating Software Development by Analysis of Changes: The Data From the Software Engineering Laboratory*, V. R. Basili and D. M. Weiss, December 1982

SEL-82-102, *FORTRAN Static Source Code Analyzer Program (SAP) System Description (Revision 1)*, W. A. Taylor and W. J. Decker, April 1985

SEL-82-105, *Glossary of Software Engineering Laboratory Terms*, T. A. Babst, M. G. Rohleder, and F. E. McGarry, October 1983

SEL-82-1306, *Annotated Bibliography of Software Engineering Laboratory Literature*, L. Morusiewicz and J. Valett, November 1994

SEL-83-001, *An Approach to Software Cost Estimation*, F. E. McGarry, G. Page, D. N. Card, et al., February 1984

SEL-83-002, *Measures and Metrics for Software Development*, D. N. Card, F. E. McGarry, G. Page, et al., March 1984

SEL-83-003, *Collected Software Engineering Papers: Volume II*, November 1983

SEL-83-007, *Proceedings of the Eighth Annual Software Engineering Workshop*, November 1983

SEL-83-106, *Monitoring Software Development Through Dynamic Variables (Revision 1)*, C. W. Doerflinger, November 1989

SEL-84-003, *Investigation of Specification Measures for the Software Engineering Laboratory (SEL)*, W. W. Agresti, V. E. Church, and F. E. McGarry, December 1984

SEL-84-004, *Proceedings of the Ninth Annual Software Engineering Workshop*, November 1984

SEL-84-101, *Manager's Handbook for Software Development (Revision 1)*, L. Landis, F. E. McGarry, S. Waligora, et al., November 1990

SEL-85-001, *A Comparison of Software Verification Techniques*, D. N. Card, R. W. Selby, Jr., F. E. McGarry, et al., April 1985

SEL-85-002, *Ada Training Evaluation and Recommendations From the Gamma Ray Observatory Ada Development Team*, R. Murphy and M. Stark, October 1985

SEL-85-003, *Collected Software Engineering Papers: Volume III*, November 1985

SEL-85-004, *Evaluations of Software Technologies: Testing, CLEANROOM, and Metrics*, R. W. Selby, Jr., and V. R. Basili, May 1985

SEL-85-005, *Software Verification and Testing*, D. N. Card, E. Edwards, F. McGarry, and C. Antle, December 1985

SEL-85-006, *Proceedings of the Tenth Annual Software Engineering Workshop*, December 1985

SEL-86-001, *Programmer's Handbook for Flight Dynamics Software Development*, R. Wood and E. Edwards, March 1986

SEL-86-002, *General Object-Oriented Software Development*, E. Seidewitz and M. Stark, August 1986

SEL-86-003, *Flight Dynamics System Software Development Environment (FDS/SDE) Tutorial*, J. Buell and P. Myers, July 1986

SEL-86-004, *Collected Software Engineering Papers: Volume IV*, November 1986

SEL-86-005, *Measuring Software Design*, D. N. Card et al., November 1986

SEL-86-006, *Proceedings of the Eleventh Annual Software Engineering Workshop*, December 1986

SEL-87-001, *Product Assurance Policies and Procedures for Flight Dynamics Software Development*, S. Perry et al., March 1987

SEL-87-002, *Ada[®] Style Guide (Version 1.1)*, E. Seidewitz et al., May 1987

SEL-87-003, *Guidelines for Applying the Composite Specification Model (CSM)*, W. W. Agresti, June 1987

SEL-87-004, *Assessing the Ada[®] Design Process and Its Implications: A Case Study*, S. Godfrey, C. Brophy, et al., July 1987

SEL-87-009, *Collected Software Engineering Papers: Volume V*, November 1987

SEL-87-010, *Proceedings of the Twelfth Annual Software Engineering Workshop*, December 1987

SEL-88-001, *System Testing of a Production Ada Project: The GRODY Study*, J. Seigle, L. Esker, and Y. Shi, November 1988

SEL-88-002, *Collected Software Engineering Papers: Volume VI*, November 1988

SEL-88-003, *Evolution of Ada Technology in the Flight Dynamics Area: Design Phase Analysis*, K. Quimby and L. Esker, December 1988

SEL-88-004, *Proceedings of the Thirteenth Annual Software Engineering Workshop*, November 1988

SEL-88-005, *Proceedings of the First NASA Ada User's Symposium*, December 1988

SEL-89-002, *Implementation of a Production Ada Project: The GRODY Study*, S. Godfrey and C. Brophy, September 1989

SEL-89-004, *Evolution of Ada Technology in the Flight Dynamics Area: Implementation/Testing Phase Analysis*, K. Quimby, L. Esker, L. Smith, M. Stark, and F. McGarry, November 1989

SEL-89-005, *Lessons Learned in the Transition to Ada From FORTRAN at NASA/Goddard*, C. Brophy, November 1989

SEL-89-006, *Collected Software Engineering Papers: Volume VII*, November 1989

SEL-89-007, *Proceedings of the Fourteenth Annual Software Engineering Workshop*, November 1989

SEL-89-008, *Proceedings of the Second NASA Ada Users' Symposium*, November 1989

SEL-89-103, *Software Management Environment (SME) Concepts and Architecture (Revision 1)*, R. Hendrick, D. Kistler, and J. Valett, September 1992

SEL-89-301, *Software Engineering Laboratory (SEL) Database Organization and User's Guide (Revision 3)*, L. Morusiewicz, December 1993

SEL-90-001, *Database Access Manager for the Software Engineering Laboratory (DAMSEL) User's Guide*, M. Buhler, K. Pumphrey, and D. Spiegel, March 1990

SEL-90-002, *The Cleanroom Case Study in the Software Engineering Laboratory: Project Description and Early Analysis*, S. Green et al., March 1990

SEL-90-003, *A Study of the Portability of an Ada System in the Software Engineering Laboratory (SEL)*, L. O. Jun and S. R. Valett, June 1990

SEL-90-004, *Gamma Ray Observatory Dynamics Simulator in Ada (GRODY) Experiment Summary*, T. McDermott and M. Stark, September 1990

SEL-90-005, *Collected Software Engineering Papers: Volume VIII*, November 1990

SEL-90-006, *Proceedings of the Fifteenth Annual Software Engineering Workshop*, November 1990

SEL-91-001, *Software Engineering Laboratory (SEL) Relationships, Models, and Management Rules*, W. Decker, R. Hendrick, and J. Valett, February 1991

SEL-91-003, *Software Engineering Laboratory (SEL) Ada Performance Study Report*, E. W. Booth and M. E. Stark, July 1991

SEL-91-004, *Software Engineering Laboratory (SEL) Cleanroom Process Model*, S. Green, November 1991

SEL-91-005, *Collected Software Engineering Papers: Volume IX*, November 1991

SEL-91-006, *Proceedings of the Sixteenth Annual Software Engineering Workshop*, December 1991

SEL-91-102, *Software Engineering Laboratory (SEL) Data and Information Policy (Revision 1)*, F. McGarry, August 1991

SEL-92-001, *Software Management Environment (SME) Installation Guide*, D. Kistler and K. Jeletic, January 1992

SEL-92-002, *Data Collection Procedures for the Software Engineering Laboratory (SEL) Database*, G. Heller, J. Valett, and M. Wild, March 1992

SEL-92-003, *Collected Software Engineering Papers: Volume X*, November 1992

SEL-92-004, *Proceedings of the Seventeenth Annual Software Engineering Workshop*, December 1992

SEL-93-001, *Collected Software Engineering Papers: Volume XI*, November 1993

SEL-93-002, *Cost and Schedule Estimation Study Report*, S. Condon, M. Regardie, M. Stark, et al., November 1993

SEL-93-003, *Proceedings of the Eighteenth Annual Software Engineering Workshop*, December 1993

SEL-94-001, *Software Management Environment (SME) Components and Algorithms*, R. Hendrick, D. Kistler, and J. Valett, February 1994

SEL-94-002, *Software Measurement Guidebook*, M. Bassman, F. McGarry, R. Pajerski, July 1994

SEL-94-003, *C Style Guide*, J. Doland and J. Valett, August 1994

SEL-94-004, *Collected Software Engineering Papers: Volume XII*, November 1994

SEL-RELATED LITERATURE

¹⁰Abd-El-Hafiz, S. K., V. R. Basili, and G. Caldiera, "Towards Automated Support for Extraction of Reusable Components," *Proceedings of the IEEE Conference on Software Maintenance-1991 (CSM 91)*, October 1991

⁴Agresti, W. W., V. E. Church, D. N. Card, and P. L. Lo, "Designing With Ada for Satellite Simulation: A Case Study," *Proceedings of the First International Symposium on Ada for the NASA Space Station*, June 1986

²Agresti, W. W., F. E. McGarry, D. N. Card, et al., "Measuring Software Technology," *Program Transformation and Programming Environments*. New York: Springer-Verlag, 1984

¹Bailey, J. W., and V. R. Basili, "A Meta-Model for Software Development Resource Expenditures," *Proceedings of the Fifth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1981

⁸Bailey, J. W., and V. R. Basili, "Software Reclamation: Improving Post-Development Reusability," *Proceedings of the Eighth Annual National Conference on Ada Technology*, March 1990

¹⁰Bailey, J. W., and V. R. Basili, "The Software-Cycle Model for Re-Engineering and Reuse," *Proceedings of the ACM Tri-Ada 91 Conference*, October 1991

¹Basili, V. R., "Models and Metrics for Software Management and Engineering," *ASME Advances in Computer Technology*, January 1980, vol. 1

Basili, V. R., *Tutorial on Models and Metrics for Software Management and Engineering*. New York: IEEE Computer Society Press, 1980 (also designated SEL-80-008)

³Basili, V. R., "Quantitative Evaluation of Software Methodology," *Proceedings of the First Pan-Pacific Computer Conference*, September 1985

⁷Basili, V. R., *Maintenance = Reuse-Oriented Software Development*, University of Maryland, Technical Report TR-2244, May 1989

⁷Basili, V. R., *Software Development: A Paradigm for the Future*, University of Maryland, Technical Report TR-2263, June 1989

⁸Basili, V. R., "Viewing Maintenance of Reuse-Oriented Software Development," *IEEE Software*, January 1990

¹Basili, V. R., and J. Beane, "Can the Parr Curve Help With Manpower Distribution and Resource Estimation Problems?," *Journal of Systems and Software*, February 1981, vol. 2, no. 1

⁹Basili, V. R., G. Caldiera, and G. Cantone, "A Reference Architecture for the Component Factory," *ACM Transactions on Software Engineering and Methodology*, January 1992

¹⁰Basili, V., G. Caldiera, F. McGarry, et al., "The Software Engineering Laboratory—An Operational Software Experience Factory," *Proceedings of the Fourteenth International Conference on Software Engineering (ICSE 92)*, May 1992

¹Basili, V. R., and K. Freburger, "Programming Measurement and Estimation in the Software Engineering Laboratory," *Journal of Systems and Software*, February 1981, vol. 2, no. 1

¹²Basili, V., and S. Green, "Software Process Evolution at the SEL," *IEEE Software*, July 1994

³Basili, V. R., and N. M. Panlilio-Yap, "Finding Relationships Between Effort and Other Variables in the SEL," *Proceedings of the International Computer Software and Applications Conference*, October 1985

⁴Basili, V. R., and D. Patnaik, *A Study on Fault Prediction and Reliability Assessment in the SEL Environment*, University of Maryland, Technical Report TR-1699, August 1986

²Basili, V. R., and B. T. Perricone, "Software Errors and Complexity: An Empirical Investigation," *Communications of the ACM*, January 1984, vol. 27, no. 1

¹Basili, V. R., and T. Phillips, "Evaluating and Comparing Software Metrics in the Software Engineering Laboratory," *Proceedings of the ACM SIGMETRICS Symposium/Workshop: Quality Metrics*, March 1981

³Basili, V. R., and C. L. Ramsey, "ARROWSMITH-P—A Prototype Expert System for Software Engineering Management," *Proceedings of the IEEE/MITRE Expert Systems in Government Symposium*, October 1985

Basili, V. R., and J. Ramsey, *Structural Coverage of Functional Testing*, University of Maryland, Technical Report TR-1442, September 1984

Basili, V. R., and R. Reiter, "Evaluating Automatable Measures for Software Development," *Proceedings of the Workshop on Quantitative Software Models for Reliability, Complexity, and Cost*. New York: IEEE Computer Society Press, 1979

⁵Basili, V. R., and H. D. Rombach, "Tailoring the Software Process to Project Goals and Environments," *Proceedings of the 9th International Conference on Software Engineering*, March 1987

⁵Basili, V. R., and H. D. Rombach, "T A M E: Tailoring an Ada Measurement Environment," *Proceedings of the Joint Ada Conference*, March 1987

⁵Basili, V. R., and H. D. Rombach, "T A M E: Integrating Measurement Into Software Environments," University of Maryland, Technical Report TR-1764, June 1987

⁶Basili, V. R., and H. D. Rombach, "The TAME Project: Towards Improvement-Oriented Software Environments," *IEEE Transactions on Software Engineering*, June 1988

⁷Basili, V. R., and H. D. Rombach, *Towards A Comprehensive Framework for Reuse: A Reuse-Enabling Software Evolution Environment*, University of Maryland, Technical Report TR-2158, December 1988

⁸Basili, V. R., and H. D. Rombach, *Towards A Comprehensive Framework for Reuse: Model-Based Reuse Characterization Schemes*, University of Maryland, Technical Report TR-2446, April 1990

⁹Basili, V. R., and H. D. Rombach, "Support for Comprehensive Reuse," *Software Engineering Journal*, September 1991

³Basili, V. R., and R. W. Selby, Jr., "Calculation and Use of an Environment's Characteristic Software Metric Set," *Proceedings of the Eighth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1985

Basili, V. R., and R. W. Selby, "Comparing the Effectiveness of Software Testing Strategies," *IEEE Transactions on Software Engineering*, December 1987

³Basili, V. R., and R. W. Selby, Jr., "Four Applications of a Software Data Collection and Analysis Methodology," *Proceedings of the NATO Advanced Study Institute*, August 1985

⁵Basili, V. R., and R. Selby, "Comparing the Effectiveness of Software Testing Strategies," *IEEE Transactions on Software Engineering*, December 1987

⁹Basili, V. R., and R. W. Selby, "Paradigms for Experimentation and Empirical Studies in Software Engineering," *Reliability Engineering and System Safety*, January 1991

⁴Basili, V. R., R. W. Selby, Jr., and D. H. Hutchens, "Experimentation in Software Engineering," *IEEE Transactions on Software Engineering*, July 1986

²Basili, V. R., R. W. Selby, and T. Phillips, "Metric Analysis and Data Validation Across FORTRAN Projects," *IEEE Transactions on Software Engineering*, November 1983

²Basili, V. R., and D. M. Weiss, *A Methodology for Collecting Valid Software Engineering Data*, University of Maryland, Technical Report TR-1235, December 1982

³Basili, V. R., and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," *IEEE Transactions on Software Engineering*, November 1984

¹Basili, V. R., and M. V. Zelkowitz, "The Software Engineering Laboratory: Objectives," *Proceedings of the Fifteenth Annual Conference on Computer Personnel Research*, August 1977

Basili, V. R., and M. V. Zelkowitz, "Designing a Software Measurement Experiment," *Proceedings of the Software Life Cycle Management Workshop*, September 1977

¹Basili, V. R., and M. V. Zelkowitz, "Operation of the Software Engineering Laboratory," *Proceedings of the Second Software Life Cycle Management Workshop*, August 1978

¹Basili, V. R., and M. V. Zelkowitz, "Measuring Software Development Characteristics in the Local Environment," *Computers and Structures*, August 1978, vol. 10

Basili, V. R., and M. V. Zelkowitz, "Analyzing Medium Scale Software Development," *Proceedings of the Third International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1978

Bassman, M. J., F. McGarry, and R. Pajerski, *Software Measurement Guidebook*, NASA-GB-001-04, Software Engineering Program, July 1994

⁹Booth, E. W., and M. E. Stark, "Designing Configurable Software: COMPASS Implementation Concepts," *Proceedings of Tri-Ada 1991*, October 1991

¹⁰Booth, E. W., and M. E. Stark, "Software Engineering Laboratory Ada Performance Study—Results and Implications," *Proceedings of the Fourth Annual NASA Ada User's Symposium*, April 1992

¹⁰Briand, L. C., and V. R. Basili, "A Classification Procedure for the Effective Management of Changes During the Maintenance Process," *Proceedings of the 1992 IEEE Conference on Software Maintenance (CSM 92)*, November 1992

¹⁰Briand, L. C., V. R. Basili, and C. J. Hetmanski, "Providing an Empirical Basis for Optimizing the Verification and Testing Phases of Software Development," *Proceedings of the Third IEEE International Symposium on Software Reliability Engineering (ISSRE 92)*, October 1992

- ¹¹Briand, L. C., V. R. Basili, and C. J. Hetmanski, *Developing Interpretable Models with Optimized Set Reduction for Identifying High Risk Software Components*, TR-3048, University of Maryland, Technical Report, March 1993
- ¹²Briand, L. C., V. R. Basili, Y. Kim, and D. R. Squier, "A Change Analysis Process to Characterize Software Maintenance Projects," *Proceedings of the International Conference on Software Maintenance*, September 1994
- ⁹Briand, L. C., V. R. Basili, and W. M. Thomas, *A Pattern Recognition Approach for Software Engineering Data Analysis*, University of Maryland, Technical Report TR-2672, May 1991
- ¹¹Briand, L. C., S. Morasca, and V. R. Basili, "Measuring and Assessing Maintainability at the End of High Level Design," *Proceedings of the 1993 IEEE Conference on Software Maintenance (CSM 93)*, November 1993
- ¹²Briand, L., S. Morasca, and V. R. Basili, *Defining and Validating High-Level Design Metrics*, University of Maryland, Technical Report TR-3301, June 1994
- ¹¹Briand, L. C., W. M. Thomas, and C. J. Hetmanski, "Modeling and Managing Risk Early in Software Development," *Proceedings of the Fifteenth International Conference on Software Engineering (ICSE 93)*, May 1993
- ⁵Brophy, C. E., W. W. Agresti, and V. R. Basili, "Lessons Learned in Use of Ada-Oriented Design Methods," *Proceedings of the Joint Ada Conference*, March 1987
- ⁶Brophy, C. E., S. Godfrey, W. W. Agresti, and V. R. Basili, "Lessons Learned in the Implementation Phase of a Large Ada Project," *Proceedings of the Washington Ada Technical Conference*, March 1988
- ²Card, D. N., "Early Estimation of Resource Expenditures and Program Size," Computer Sciences Corporation, Technical Memorandum, June 1982
- ²Card, D. N., "Comparison of Regression Modeling Techniques for Resource Estimation," Computer Sciences Corporation, Technical Memorandum, November 1982
- ³Card, D. N., "A Software Technology Evaluation Program," *Anais do XVIII Congresso Nacional de Informatica*, October 1985
- ⁵Card, D. N., and W. W. Agresti, "Resolving the Software Science Anomaly," *Journal of Systems and Software*, 1987
- ⁶Card, D. N., and W. W. Agresti, "Measuring Software Design Complexity," *Journal of Systems and Software*, June 1988
- ⁴Card, D. N., V. E. Church, and W. W. Agresti, "An Empirical Study of Software Design Practices," *IEEE Transactions on Software Engineering*, February 1986
- Card, D. N., V. E. Church, W. W. Agresti, and Q. L. Jordan, "A Software Engineering View of Flight Dynamics Analysis System," Parts I and II, Computer Sciences Corporation, Technical Memorandum, February 1984

Card, D. N., Q. L. Jordan, and V. E. Church, "Characteristics of FORTRAN Modules," Computer Sciences Corporation, Technical Memorandum, June 1984

⁵Card, D. N., F. E. McGarry, and G. T. Page, "Evaluating Software Engineering Technologies," *IEEE Transactions on Software Engineering*, July 1987

³Card, D. N., G. T. Page, and F. E. McGarry, "Criteria for Software Modularization," *Proceedings of the Eighth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1985

¹Chen, E., and M. V. Zelkowitz, "Use of Cluster Analysis To Evaluate Software Engineering Methodologies," *Proceedings of the Fifth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1981

⁴Church, V. E., D. N. Card, W. W. Agresti, and Q. L. Jordan, "An Approach for Assessing Software Prototypes," *ACM Software Engineering Notes*, July 1986

²Doerflinger, C. W., and V. R. Basili, "Monitoring Software Development Through Dynamic Variables," *Proceedings of the Seventh International Computer Software and Applications Conference*. New York: IEEE Computer Society Press, 1983

Doubleday, D., *ASAP: An Ada Static Source Code Analyzer Program*, University of Maryland, Technical Report TR-1895, August 1987 (NOTE: 100 pages long)

⁶Godfrey, S., and C. Brophy, "Experiences in the Implementation of a Large Ada Project," *Proceedings of the 1988 Washington Ada Symposium*, June 1988

⁵Jeffery, D. R., and V. Basili, *Characterizing Resource Data: A Model for Logical Association of Software Data*, University of Maryland, Technical Report TR-1848, May 1987

⁶Jeffery, D. R., and V. R. Basili, "Validating the TAME Resource Data Model," *Proceedings of the Tenth International Conference on Software Engineering*, April 1988

¹¹Li, N. R., and M. V. Zelkowitz, "An Information Model for Use in Software Management Estimation and Prediction," *Proceedings of the Second International Conference on Information Knowledge Management*, November 1993

⁵Mark, L., and H. D. Rombach, *A Meta Information Base for Software Engineering*, University of Maryland, Technical Report TR-1765, July 1987

⁶Mark, L., and H. D. Rombach, "Generating Customized Software Engineering Information Bases From Software Process and Product Specifications," *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, January 1989

⁵McGarry, F. E., and W. W. Agresti, "Measuring Ada for Software Development in the Software Engineering Laboratory (SEL)," *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, January 1988

⁷McGarry, F., L. Esker, and K. Quimby, "Evolution of Ada Technology in a Production Software Environment," *Proceedings of the Sixth Washington Ada Symposium (WADAS)*, June 1989

³McGarry, F. E., J. Valett, and D. Hall, "Measuring the Impact of Computer Resource Quality on the Software Development Process and Product," *Proceedings of the Hawaiian International Conference on System Sciences*, January 1985

³Page, G., F. E. McGarry, and D. N. Card, "A Practical Experience With Independent Verification and Validation," *Proceedings of the Eighth International Computer Software and Applications Conference*, November 1984

¹²Porter, A. A., L. G. Votta, Jr., and V. R. Basili, *Comparing Detection Methods for Software Requirements Inspections: A Replicated Experiment*, University of Maryland, Technical Report TR-3327, July 1994

⁵Ramsey, C. L., and V. R. Basili, "An Evaluation of Expert Systems for Software Engineering Management," *IEEE Transactions on Software Engineering*, June 1989

³Ramsey, J., and V. R. Basili, "Analyzing the Test Process Using Structural Coverage," *Proceedings of the Eighth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1985

⁵Rombach, H. D., "A Controlled Experiment on the Impact of Software Structure on Maintainability," *IEEE Transactions on Software Engineering*, March 1987

⁸Rombach, H. D., "Design Measurement: Some Lessons Learned," *IEEE Software*, March 1990

⁹Rombach, H. D., "Software Reuse: A Key to the Maintenance Problem," *Butterworth Journal of Information and Software Technology*, January/February 1991

⁶Rombach, H. D., and V. R. Basili, "Quantitative Assessment of Maintenance: An Industrial Case Study," *Proceedings From the Conference on Software Maintenance*, September 1987

⁶Rombach, H. D., and L. Mark, "Software Process and Product Specifications: A Basis for Generating Customized SE Information Bases," *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, January 1989

⁷Rombach, H. D., and B. T. Ulery, *Establishing a Measurement Based Maintenance Improvement Program: Lessons Learned in the SEL*, University of Maryland, Technical Report TR-2252, May 1989

¹⁰Rombach, H. D., B. T. Ulery, and J. D. Valett, "Toward Full Life Cycle Control: Adding Maintenance Measurement to the SEL," *Journal of Systems and Software*, May 1992

⁶Seidewitz, E., "Object-Oriented Programming in Smalltalk and Ada," *Proceedings of the 1987 Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1987

⁵Seidewitz, E., "General Object-Oriented Software Development: Background and Experience," *Proceedings of the 21st Hawaii International Conference on System Sciences*, January 1988

⁶Seidewitz, E., "General Object-Oriented Software Development with Ada: A Life Cycle Approach," *Proceedings of the CASE Technology Conference*, April 1988

⁹Seidewitz, E., "Object-Oriented Programming Through Type Extension in Ada 9X," *Ada Letters*, March/April 1991

¹⁰Seidewitz, E., "Object-Oriented Programming With Mixins in Ada," *Ada Letters*, March/April 1992

¹²Seidewitz, E., "Genericity Versus Inheritance Reconsidered: Self-Reference Using Generics," *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1994

⁴Seidewitz, E., and M. Stark, "Towards a General Object-Oriented Software Development Methodology," *Proceedings of the First International Symposium on Ada for the NASA Space Station*, June 1986

⁹Seidewitz, E., and M. Stark, "An Object-Oriented Approach to Parameterized Software in Ada," *Proceedings of the Eighth Washington Ada Symposium*, June 1991

⁸Stark, M., "On Designing Parametrized Systems Using Ada," *Proceedings of the Seventh Washington Ada Symposium*, June 1990

¹¹Stark, M., "Impacts of Object-Oriented Technologies: Seven Years of SEL Studies," *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, September 1993

⁷Stark, M. E. and E. W. Booth, "Using Ada to Maximize Verbatim Software Reuse," *Proceedings of TRI-Ada 1989*, October 1989

⁵Stark, M., and E. Seidewitz, "Towards a General Object-Oriented Ada Lifecycle," *Proceedings of the Joint Ada Conference*, March 1987

¹⁰Straub, P. A., and M. V. Zelkowitz, "On the Nature of Bias and Defects in the Software Specification Process," *Proceedings of the Sixteenth International Computer Software and Applications Conference (COMPSAC 92)*, September 1992

⁸Straub, P. A., and M. V. Zelkowitz, "PUC: A Functional Specification Language for Ada," *Proceedings of the Tenth International Conference of the Chilean Computer Science Society*, July 1990

⁷Sunazuka, T., and V. R. Basili, *Integrating Automated Support for a Software Management Cycle Into the TAME System*, University of Maryland, Technical Report TR-2289, July 1989

¹⁰Tian, J., A. Porter, and M. V. Zelkowitz, "An Improved Classification Tree Analysis of High Cost Modules Based Upon an Axiomatic Definition of Complexity," *Proceedings of the Third IEEE International Symposium on Software Reliability Engineering (ISSRE 92)*, October 1992

Turner, C., and G. Caron, *A Comparison of RADC and NASA/SEL Software Development Data*, Data and Analysis Center for Software, Special Publication, May 1981

¹⁰Valett, J. D., "Automated Support for Experience-Based Software Management," *Proceedings of the Second Irvine Software Symposium (ISS '92)*, March 1992

⁵Valett, J. D., and F. E. McGarry, "A Summary of Software Measurement Experiences in the Software Engineering Laboratory," *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, January 1988

³Weiss, D. M., and V. R. Basili, "Evaluating Software Development by Analysis of Changes: Some Data From the Software Engineering Laboratory," *IEEE Transactions on Software Engineering*, February 1985

⁵Wu, L., V. R. Basili, and K. Reed, "A Structure Coverage Tool for Ada Software Systems," *Proceedings of the Joint Ada Conference*, March 1987

¹Zelkowitz, M. V., "Resource Estimation for Medium-Scale Software Projects," *Proceedings of the Twelfth Conference on the Interface of Statistics and Computer Science*. New York: IEEE Computer Society Press, 1979

²Zelkowitz, M. V., "Data Collection and Evaluation for Experimental Computer Science Research," *Empirical Foundations for Computer and Information Science* (Proceedings), November 1982

⁶Zelkowitz, M. V., "The Effectiveness of Software Prototyping: A Case Study," *Proceedings of the 26th Annual Technical Symposium of the Washington, D. C., Chapter of the ACM*, June 1987

⁶Zelkowitz, M. V., "Resource Utilization During Software Development," *Journal of Systems and Software*, 1988

⁸Zelkowitz, M. V., "Evolution Towards Specifications Environment: Experiences With Syntax Editors," *Information and Software Technology*, April 1990

NOTES:

¹This article also appears in SEL-82-004, *Collected Software Engineering Papers: Volume I*, July 1982.

²This article also appears in SEL-83-003, *Collected Software Engineering Papers: Volume II*, November 1983.

³This article also appears in SEL-85-003, *Collected Software Engineering Papers: Volume III*, November 1985.

⁴This article also appears in SEL-86-004, *Collected Software Engineering Papers: Volume IV*, November 1986.

⁵This article also appears in SEL-87-009, *Collected Software Engineering Papers: Volume V*, November 1987.

⁶This article also appears in SEL-88-002, *Collected Software Engineering Papers: Volume VI*, November 1988.

⁷This article also appears in SEL-89-006, *Collected Software Engineering Papers: Volume VII*, November 1989.

⁸This article also appears in SEL-90-005, *Collected Software Engineering Papers: Volume VIII*, November 1990.

⁹This article also appears in SEL-91-005, *Collected Software Engineering Papers: Volume IX*, November 1991.

¹⁰This article also appears in SEL-92-003, *Collected Software Engineering Papers: Volume X*, November 1992.

¹¹This article also appears in SEL-93-001, *Collected Software Engineering Papers: Volume XI*, November 1993.

¹²This article also appears in SEL-94-004, *Collected Software Engineering Papers: Volume XII*, November 1994.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE November 1994	3. REPORT TYPE AND DATES COVERED Contractor Report	
4. TITLE AND SUBTITLE Collected Software Engineering Papers: Volume XII			5. FUNDING NUMBERS 552 <i>IN 60 7 11</i> <i>3310 S</i> <i>p- 117</i>	
6. AUTHOR(S) Software Engineering Laboratory				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Software Engineering Branch Code 552 Goddard Space Flight Center Greenbelt, Maryland			8. PERFORMING ORGANIZATION REPORT NUMBER SEL-94-004	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) NASA Aeronautics and Space Administration Washington, D.C. 20546-0001			10. SPONSORING/MONITORING AGENCY REPORT NUMBER CR-189409	
11. SUPPLEMENTARY NOTES				
12a. DISTRIBUTION/AVAILABILITY STATEMENT Unclassified-Unlimited Subject Category: 61 Report is available from the NASA Center for AeroSpace Information, 800 Elkridge Landing Road, Linthicum Heights, MD 21090; (301) 621-0390.			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) This document is a collection of selected technical papers produced by participants in the Software Engineering Laboratory (SEL) from November 1993 through October 1994. The purpose of the document is to make available, in one reference, some results of SEL research that originally appeared in a number of different forums. This is the 12th such volume of technical papers produced by the SEL. Although these papers cover several topics related to software engineering, they do not encompass the entire scope of SEL activities and interests. Additional information about the SEL and its research efforts may be obtained from the sources listed in the bibliography at the end of this document.				
14. SUBJECT TERMS Software Engineering, Software Measurement, Ada Technology, Bibliography			15. NUMBER OF PAGES 100	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT Unlimited	

